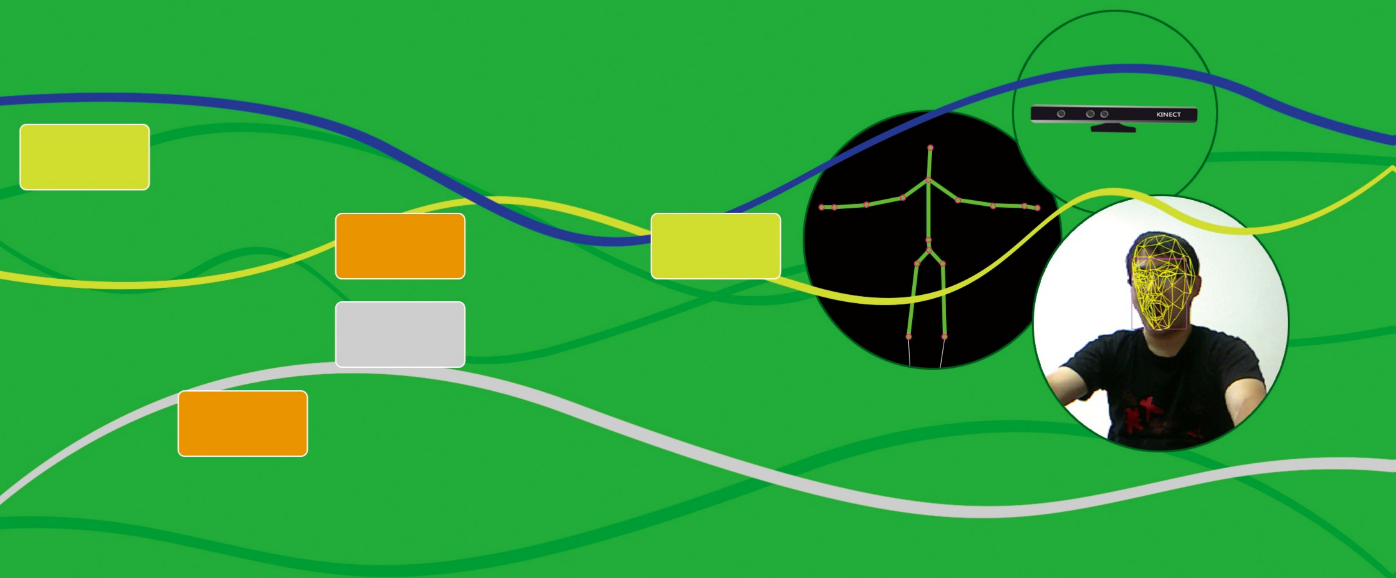



图灵原创



# KINECT人机交互开发实践

吴国斌 李斌 阎骥洲/编著






人民邮电出版社
POSTS & TELECOM PRESS



**吴国斌**

博士，PMP，微软亚洲研究院学术合作经理，负责中国高校及科研机构Kinect for Windows学术合作计划及微软精英大挑战Kinect主题项目，曾担任微软TechEd 2011 Kinect论坛讲师、微软亚洲教育高峰会Kinect分论坛主席、中国计算机学会学科前沿讲习班Kinect主题学术主任。



**李　斌**

来自西安电子科技大学，国内首批Kinect开发者，策划执行微软Kinect for Windows Pioneer计划，开发Kinect风筝项目，曾担任微软TechEd 2011 Kinect论坛讲师、人大附中Kinect选修课讲师。



**阎骥洲**

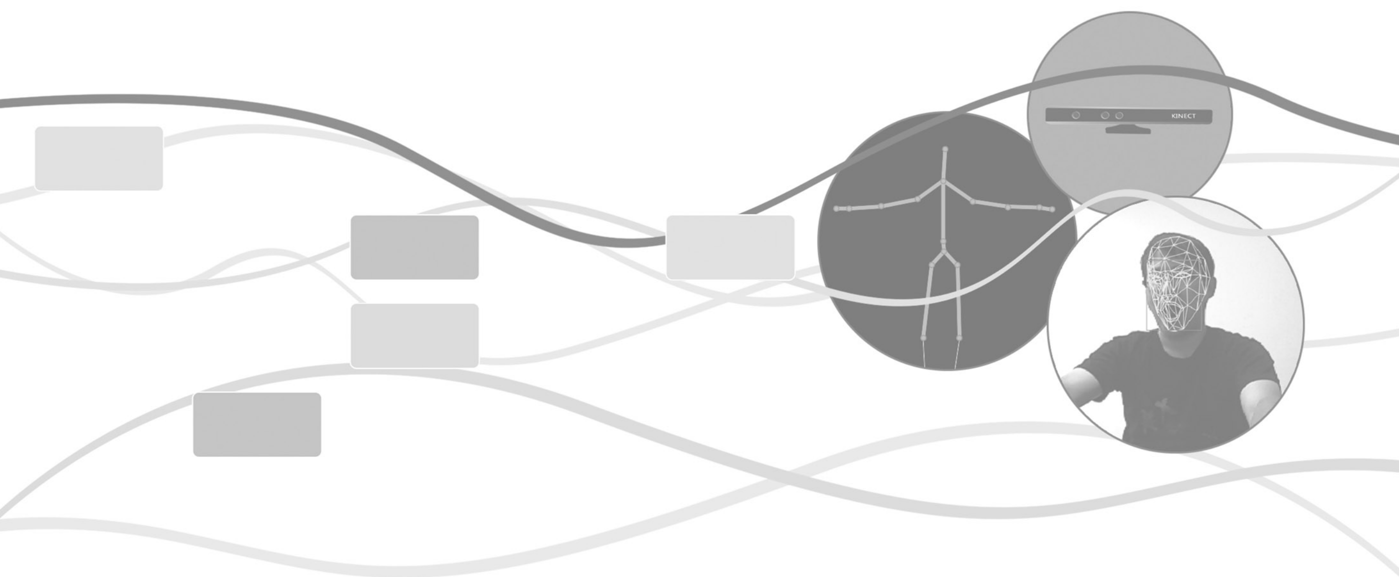
来自北京航空航天大学，国内首批Kinect开发者，曾在微软Kinect for Windows Pioneer计划中凭借“虚拟演示系统”获得第一名的成绩，之后参与并指导了多个Kinect应用开发项目，有着丰富的Kinect使用及开发经验。




图灵原创

# KINECT 人机交互开发实践

吴国斌 李斌 阎骥洲/编著




人民邮电出版社
北京

图书在版编目（ＣＩＰ）数据

Kinect人机交互开发实践 / 吴国斌，李斌，阎骥洲
编著. -- 北京 : 人民邮电出版社，2013.1
（图灵原创）
ISBN 978-7-115-30029-4

Ⅰ. ①K… Ⅱ. ①吴… ②李… ③阎… Ⅲ. ①人-机系
统－设计 Ⅳ. ①TB18

中国版本图书馆CIP数据核字(2012)第273598号

## 内 容 提 要

Kinect 是微软公司推出的最新的基于体感交互的人机交互设备。本书分为 3 个部分，首先介绍了 Kinect 的结构和功能以及如何配置相关的开发环境，接着结合实例介绍如何使用 Kinect for Windows SDK 提供的 API，最后通过 4 个实例详细讲述了使用 Kinect for Windows SDK 开发项目的实现过程。

本书旨在为 Kinect for Windows 开发人员提供快速入门的知识，但是要求读者有一定的编程基础。由于本书的实例代码全部由 C# 编写，读者最好对 C# 有一定的了解。

图灵原创
Kinect人机交互开发实践

◆ 编　　著　吴国斌　李　斌　阎骥洲
　责任编辑　王军花
　执行编辑　赵慧明
◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号
　邮编　100061　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　印刷
◆ 开本：800×1000　1/16
　印张：10.75
　字数：254千字　　　　2013年 1 月第 1 版
　印数：1－4 000册　　　2013年 1 月北京第 1 次印刷

ISBN 978-7-115-30029-4

定价：39.00元

**读者服务热线：(010)51095186转604　印装质量热线：(010)67129223**

**反盗版热线：(010)67171154**

# 序

微软从未停止过创新的步伐，Kinect的问世无疑又为自然用户界面领域打开了一片新的天地。Kinect集强大的机器学习技术、身份识别能力以及语音识别功能于一身，俨然成为了即将改变世界的又一大利器。

Kinect最先与Xbox 360结合，把Xbox 360变成了一台体感游戏机，抛开游戏手柄，你的身体就是遥控器。Xbox 360与Kinect套装开售仅3个月，便售出上千万台，创造了新的电子产品吉尼斯销售记录。

Kinect for Windows SDK的发布更是在科研工作者和开发者社区中掀起了一波创新热潮，越来越多的技术爱好者投身于探索Kinect带来的无限可能中。短短一年多的时间，我们便看到Kinect已经在各个领域大展身手。在娱乐游戏方面，有隔空切西瓜的Kinect版水果忍者；在传统文化方面，有随动皮影戏、川剧变脸和虚拟放风筝；在机械控制领域，有徒手控制直升机、机器人随动和自主避障车等；在医疗领域，手术中医生只用手势即可控制医疗影像的播放和缩放，实现了便捷且无菌的操作；在辅助工具方面，有PPT播放和虚拟演示系统等，Kinect的体感交互提供了一种全新的演讲和展示方式。由此可见，Kinect把多少不可能变成了现实，然而这只是一部分，Kinect蕴藏的无限潜力远不止这些，等待着读者们去发掘和创造。

Kinect for Windows硬件设备在2012年10月初登陆中国市场，这本Kinect开发入门教程可以说是应时而生，非常值得Kinect初学者参考。

微软亚洲研究院多个研究小组的研究成果已经转化到Kinect的核心技术中。本书的一位作者吴国斌博士在微软亚洲研究院负责Kinect for Windows在中国的学术合作计划，见证了Kinect for Windows在中国高校和科研机构的创新发展历程。另外两位作者作为第一批使用Kinect for Windows进行项目开发的技术爱好者，一直跟踪着Kinect for Windows SDK最新技术动向。我很高兴看到他们能够将自己积累的经验写成书，与更多的科研工作者和技术爱好者分享。

最后，期待在层出不穷的Kinect创新应用中看到你的作品！

郭百宁，微软亚洲研究院常务副院长

2012年10月

# 前　　言

Kinect是微软公司推出的最新的基于体感交互的人机交互设备。Kinect最初作为Xbox 360的外接设备发布，利用即时动态骨骼追踪、影像识别、麦克风输入、语音识别等功能让玩家摆脱传统游戏手柄的束缚，通过自己的肢体动作来控制游戏。到2011年3月，Kinect已经售出了1000多万台，创造了新的销售记录，同时也表明了Kinect全新的体感交互体验征服了众多玩家的心。微软公司并没有将这一先进技术局限在游戏行业，而是紧接着将Kinect技术推广到Windows平台，开放了本书要介绍的Kinect for Windows SDK，旨在鼓励众多开发者设计基于Kinect体感交互技术的应用，从而在各个行业领域里改变人们工作、生活和娱乐的方式。

## 本书结构

本书分为3个部分，具体如下所示。

- Kinect基础篇。介绍Kinect的结构和功能以及如何配置开发环境，主要包含2章内容。
  - 第1章简要介绍Kinect的概念、历史、结构组成及其应用领域。
  - 第2章将一步一步带领读者进行Kinect for Windows开发环境的配置。
- Kinect开发篇。结合实例介绍如何使用Kinect for Windows SDK提供的API，主要包含6章内容。
  - 第3章介绍Kinect获取的彩色图像数据和红外图像数据，并结合实例介绍如何调用Kinect for Windows SDK提供的API获取这两种图像数据。
  - 第4章介绍Kinect获取的深度图像数据，并结合实例介绍如何处理深度图像数据。
  - 第5章介绍骨骼追踪数据，包括其结构、半身模式以及新加的骨骼点旋转信息，通过实例3讲解骨骼数据API的调用方法，通过实例4介绍如何利用骨骼追踪数据实现相应的功能。
  - 第6章介绍Kinect for Windows SDK中音频API的使用方法，实例5和实例6分别讲解了如何记录Kinect捕获到的音频流以及如何进行语音识别。
  - 第7章介绍Kinect for Windows Developer Toolkit，包括如何对其进行安装，如何利用其中的Kinect Studio进行便捷开发，以及Face Tracking SDK， 并通过实例7详细讲解如何使用Face Tracking SDK识别人脸。
  - 第8章讲述了Kinect常用的两个类库：Coding4Fun Kinect Toolkit和Kinect Toolbox。
- Kinect实战篇。通过4个实例详细讲述了使用Kinect for Windows SDK开发项目的实现过程，

包括4章内容。

- 第9章讲解了Kinect虚拟演示系统的实现过程，该项目结合Kinect提供了一种新颖的演讲方式。
- 第10章讲解了Kinect虚拟风筝项目，将微软最新的Kinect姿势识别技术与风筝文化结合，提供一种新的虚拟放风筝体验。
- 第11章介绍了虚拟博物馆的实现。该项目利用Kinect SDK 提供的骨骼点追踪功能，结合普通的显示屏或者投影仪实现了全息显示的效果。
- 第12章讲述了基于Kinect的自主移动机器人项目。本项目将Kinect作为机器人的视觉传感器，指导机器人移动。

本书内容丰富，通过对基础篇和开发篇的学习，读者可以了解Kinect技术的相关知识，并掌握Kinect相关API的使用方法。其中开发篇附带了大量的示例程序，希望读者能够亲自试验。实战篇讲解了多个Kinect相关项目，希望能够对读者在开发过程中提供一定的参考。

## 读者对象

本书旨在为Kinect for Windows开发人员提供快速入门的知识，但是要求读者有一定的编程基础。由于本书的实例代码全部由C#编写，读者最好对C#有一定的了解。

## 致谢

本书的出版得了各方面的帮助和支持。感谢微软亚洲研究院副院长宋罗兰女士和资深学术合作经理马歆女士、Miran Lee女士对本书写作给予的大力支持。感谢美国总部微软研究院Stewart Tansley博士、Kinect for Windows部门Bob Heddle、Anson Tsao、李杭和Heather Mitchell给予的指导。感谢清华大学美术学院徐迎庆教授和他的学生赵月、张爽爽、于阳、魏一明，实战部分中的虚拟风筝项目是在徐老师指导下，由4位同学完成交互界面的设计。感谢微软技术俱乐部北京航空航天大学虚拟演示项目组张蕊、伍彦飞同学和虚拟博物馆项目组黎健成、范若余同学，他们的创意项目很大程度上丰富了书中实战部分的内容。感谢西安交通大学的袁博沣、姚佳文和赵文玉同学，书中Kinect机器人项目就是由他们完成的。此外，感谢微软亚太研发集团Kinect for Windows部门王维，中国科学技术大学李明磊和北京航空航天大学陈璇在百忙中审阅部分文稿。再次感谢所有为本书编写和出版付出辛勤劳动的同仁。

# 目　　录

## 第一部分　Kinect基础篇



## 第二部分　Kinect开发篇

# 第三部分 Kinect实战篇

*Part 1*

第一部分

# Kinect 基础篇

2011 年 3 月 9 日，微软宣布 Kinect 自发售以来已经售出了 1000 多万部，销售额超过 15 亿美元。与此同时，他们还售出了 1000 多万套专为 Kinect 设计的游戏。根据吉尼斯世界记录，Kinect 成为了历史上销售速度最快的消费类电子产品，打破了此前由 iPhone 和 iPad 所保持的记录。微软 Kinect 以“你就是主角”（You are the controller）为口号，正在引领着一场人机交互的变革。

进行 Kinect for Windows 开发，首先要配置开发环境。本书将在第 2 章详细介绍 Kinect 开发对软硬件的要求，并带领读者一步一步进行环境的配置。这部分最后还会结合自然用户界面（NUI），简要介绍 Kinect for Windows SDK 的整体框架。

第 1 章

# Kinect简介

1

Kinect被誉为第三代人机交互的划时代之作。本章将介绍Kinect的基本概念及其发展历程，并简要剖析其结构功能以及体感交互技术的原理。另外，本章最后还会对Kinect for Windows的应用领域进行概览和展望。

## 1.1 什么是 Kinect

Kinect是Xbox 360外接的3D体感摄影机，如图1-1所示。它利用即时动态捕捉、影像识别、麦克风输入、语音识别等功能，使玩家摆脱了传统游戏手柄的束缚，使用自己的肢体来控制游戏。而任天堂Wii、索尼Play Station Move等同类产品，则需要玩家借助一个或者多个设备才能完成体感互动。




图1-1 Kinect for Xbox 360

作为Xbox 360的外设，Kinect不需要使用任何道具即可完成整个动作的识别和捕捉，它使用了由微软剑桥研究院研发的基于深度图像的人体骨骼追踪算法，而深度图像则是由PrimeSense公司提供的Range Camera技术产生的。此外，Kinect使用一个4-麦克阵列，可以识别3D立体语音。

Kinect的主要识别算法和软件部分都是由微软旗下的游戏工作室提供的。国内外一些所谓的可见光或者红外识别公司，大多是从该工作室获取一些专利权，其产品跟微软的Kinect相比在精度上还有一定的差距。

## 1.2 Kinect 的前世今生

2009年6月1日，Kinect在E3游戏展上首次亮相，它当时的代号是Project Natal。这遵循了微软以城市名作为开发代号的传统，Project Natal是由来自巴西的微软董事Alex Kipman以巴西城市Natal命名的。Natal是拉丁语，英语中有“初生”之意，由此可见，微软公司期望Kinect能够给Xbox 360带来新生。在E3 2009游戏展上，Kinect的骨骼捕捉技术已经可以在30Hz的条件下同时捕捉4个人的48个骨骼动作。

2010年3月25日，微软宣布将在E3 2010期间召开的“初生计划全球首秀”发布会上公布Kincet的发售日期。2010年6月13日晚，这个发布会在格兰中心体育馆举行，会上微软宣布将Project Natal正式命名为Kinect，这融合了kinetic（运动）和connect（沟通）之意。同时微软还宣布，Kinect将于2010年11月4日在北美正式发售。

Kinect在发布仅仅两个月后，就售出了800多万台，吉尼斯世界记录称其为有史以来销售最快的电子消费产品。但是，Kinect并未就此止步。2011年6月，Kinect for Windows SDK beta版发布，这标志着Kinect开始向PC应用领域进军。2011年11月4日，Kinect发布一周年的日子，世界各地的研究人员已经将Kinect应用到了医疗健康、教育、日常生活等各个领域，以探索Kinect技术的无限可能，这就是所谓的“Kinect效应”。此外，Kinect动作捕捉的机器学习技术还荣获了2011年MacRobert Award工程创新大奖。

微软在Kinect for Xbox 360设备的基础上优化了硬件组件，并于2012年2月发布了Kinect for Windows硬件，其固件更适合PC使用。新的Kinect硬件缩短了USB连接线的长度，并支持“近距模式”（Near Mode）。与此同时，微软还发布了商业授权版的Kinect for Windows SDK 1.0，这意味着开发者可以使用Kinect for Windows硬件，在Windows平台上开发支持手势和语音识别的应用程序，并向实际用户销售这些程序。

对于商业版的Kinect for Windows，微软采用了纯硬件的商业模式，向开发人员和软件商免费提供SDK开发包。这样，所有的使用者都可以将精力投入到研发上，而不必担心支付任何软件的授权费用。

2012年5月，微软发布了Kinect for Windows SDK 1.5版本，该版本支持人脸以及坐姿半身模式的骨骼追踪。借助这些新功能和特性，Kinect应用程序的开发工作变得更加容易和灵活。2012年10月，微软又发布了Kinect for Windows SDK 1.6版本，主要拓展了Kinect for Windows的开发平台，支持在虚拟机、Windows 8系统上进行开发，支持使用最新的Visual Studio 2012开发工具。此外，Kinect for Windows SDK 1.6版本还增加了获取红外图像等功能，并在性能上做了很大提升。Kinect for Windows SDK可能会保持每年一到两次的更新，在功能和性能上也会越来越强大。

## 1.3　Kinect 体感交互技术原理

初看Kinect，你或许只看到了3个小摄像头，那么Kinect究竟是怎样实现体感交互的呢？Kinect for Windows SDK又有哪些基本功能呢？本节将揭开Kinect在硬件、软件方面的神秘面纱。

### 1.3.1　Kinect的结构组成

图1-2给出了Kinect的整体结构。Kinect一共有3个摄像头，中间一个是RGB摄像头，用来获取640×480的彩色图像，每秒钟最多获取30帧图像；两边的是两个深度传感器，左侧的是红外线发射器，右侧的是红外线接收器，用来检测玩家的相对位置。Kinect的两侧是一组四元麦克风阵列，用于声源定位和语音识别；下方还有一个带内置马达的底座，可以调整俯仰角。




图1-2　Kinect硬件结构

### 1.3.2　Kinect for Windows SDK 简介及功能介绍

2011年6月17日，微软研究院发布的非商业授权版的Kinect for Windows SDK Beta吸引了众多开发者的目光，不过该版本只允许用于研究、测试和实验，不可以发布商业应用。2012年，微软分别在2月、5月和10月接连发布了商业授权版的Kinect for Windows SDK 1.0版本、1.5版本和1.6版本，此举在明确了微软盈利模式的同时，使得开发者可以进行软件开发，并销售开发的应用程序。

Kinect for Windows SDK目前支持Windows 7操作系统和Windows 8操作系统，开发环境使用Visual Studio 2010 Express及以上版本，支持的开发语言包括C++、C#和VB.NET。

Kinect for Windows SDK主要包括以下几个功能。

- 骨骼追踪：对在Kinect视野范围内移动的一个或两个人进行骨骼追踪，可以追踪到人体上的20个节点。此外，Kinect还支持更精确的人脸识别。
- 深度摄像头：利用“光编码”技术，通过深度传感器获取到视野内的环境三维位置信息。这种深度数据可以简单地理解为一张利用特殊摄像头获取到的图像，但是其每一个像素

的数据不是普通彩色图片的像素值，而是这个像素的位置距离Kinect传感器的距离。由于这种技术是利用Kinect红外发射器发出的红外线对空间进行编码的，因此无论环境光线如何，测量结果都不会受到干扰。

- 音频处理：与Microsoft Speech的语音识别API集成，使用一组具有消除噪声和回波的四元麦克风阵列，能够捕捉到声源附近有效范围之内的各种信息。

## 1.4 Kinect for Windows 应用领域

目前，国外已经出现了很多使用Kinect开发的精彩应用，比如Kinect试衣镜、Air Presenter演讲软件、Kinect光剑、Kinect街头霸王等。很多创意都可以在MSDN Channel 9的Coding4fun栏目里看到。

在国内，Kinect for Windows SDK Beta发布伊始，微软亚洲研究院便启动了“微软校园菁英计划”之Kinect Pioneer项目，在全国范围内动员微软学生技术俱乐部的同学们集思广益，提交他们基于Kinect的新创意，并向优秀的创意团队提供Kinect设备和技术支持。仅一个多月的开发时间，多个优秀创意团队便成功提交了Kinect创意项目原型，其中包括使用手势进行变脸的3D脸谱虚拟平台、Kinect教学助手、基于Kinect的网上试衣间等。在随后的2012微软精英大挑战Kinect主题上，来自全国30所高校的100余支队伍也积极参与到Kinect for Windows的开发当中，这使得Kinect在中国大学生中得到了全面的推广。

下面简单介绍一下来自西安电子科技大学团队的3D脸谱虚拟平台。此创意将京剧这门传统艺术和新颖的Kinect技术结合到了一起，通过Kinect搭建了一个可以让京剧迷享受虚拟演唱体验的平台：一个提供脸谱、服装和场景的华丽舞台。凭借着Kinect的人体识别和传感技术，用户可以直接跳过繁复的化妆过程，用手势来选择自己喜爱的角色脸谱，生、旦、净、末、丑，一应俱全，选择完毕后，就可以对着屏幕表演喜爱的曲目并录像了，如图1-3所示。当与其他戏迷朋友分享视频时，他们可以欣赏到惟妙惟肖的场景和表演。这样的方式不仅能使老京剧迷们的交流更加便捷，还能让年轻人通过更炫的途径去了解这门生动的国粹。



图1-3 3D脸谱虚拟平台

2011年12月2日，由微软亚洲研究院举办的Kinect for Windows研讨会在北京召开，吸引了来自众多行业和研究领域的专家学者以及全国各地的大学老师和学生。研讨会就Kinect的体感交互技术及其应用领域进行了交流和讨论，并且展示了国内基于Kinect for Windows SDK在各个应用领域的研发成果。下面选择一些有代表性的项目进行简要介绍。

- 基于Kinect的手语翻译系统。手语翻译系统旨在解决聋哑人与正常人的沟通问题，利用Kinect对肢体动作的实时捕捉，对特定的手语动作进行识别，最终翻译成文字，这样不懂手语的人也可以跟聋哑人正常交流了。目前，该项目的研究已经取得了重大的进展，我们相信结合Kinect强大的体感交互技术，在不远的将来就会看到Kinect手语翻译系统成为聋哑人的得力助手。
- 空中手写。空中手写软件巧妙地利用了Kinect对手部节点的实时追踪，用户只需对着Kinect在空气中比划，便能写出相应的汉字，并输入到计算机中。该项目为未来的手写输入提供了新的思路。
- 虚拟试衣系统。虚拟试衣系统实现了不需用户真正穿上衣服便可看到衣服上身效果的虚拟试衣体验。用户只需站在屏幕前，选择虚拟试衣系统中存储的品牌衣服图片，系统会根据Kinect获取到的骨架数据自适应地穿在用户在屏幕中的影像上，达到轻轻松松试衣的效果。这样不仅使得试衣变得更加便捷、有趣，还能有效减少试衣成本。
- Kinect版水果忍者。Kinect版水果忍者也是利用了Kinect的骨骼追踪技术，将触屏版的水果忍者游戏移植到Kinect的体感控制上。Kinect追踪玩家双手的移动轨迹，玩家只需对着屏幕划动双手便可切下水果，俨然真实版忍者，在娱乐的同时还能锻炼身体。

## 1.5　小结

Kinect是一种廉价的动作捕捉设备，适用于对动作捕捉精度要求非常严格的领域，这也是其未来发展的方向。另外，现在智能手机和平板电脑的发展非常迅猛，虽然现在看它们和PC会怎样发展，还没有定论，但是受此趋势影响，Kinect以后绝对会趋于小型化，可以断定这是其发展的必经之路。大家都知道，专业领域的产品用量通常不会很大，但是这个领域的技术更新相对较快，随着技术的完善，最终专业领域的技术一定会逐渐应用到消费者领域。互联网的发展历史就是很好的明证，其他许多成功普及的技术也都遵循着这样的发展轨迹。

第 2 章

# Kinect for Windows开发环境配置

工欲善其事，必先利其器。要进行Kinect for Windows开发，首先需要配置相应的开发环境。本章将详细介绍开发环境的软硬件需求，并带领大家一步一步配置开发环境，此外，还会简要介绍NUI API所包含的内容。

## 2.1 开发环境需求

Kinect的开发环境对计算机的软硬件有一定的要求，但并不高，目前主流的计算机配置大都能满足其需求。

### 1. 硬件需求

计算机硬件方面的要求如下：

- 需要拥有双核、2.66GHz以上的CPU；
- 显卡支持Microsoft DirectX 9.0c；
- 2GB的RAM；
- Kinect for Windows传感器以及一根专用的电源适配线。

### 2. 系统需求

目前，Kinect for Windows SDK支持Windows 7操作系统和Windows 8操作系统。另外，Kinect for Windows SDK还支持虚拟机，可以使用的虚拟机环境有Microsoft HyperV、VMWare和Parallels。

### 3. 软件需求

除了硬件需求和系统需求外，Kinect for Windows开发还需要配备有以下环境。

- Microsoft Visual Studio 2010 或Visual Studio 2012，本书使用Microsoft Visual Studio 2010（它可以从微软官网免费下载，网址是http://www.microsoft.com/visualstudio/en-us/products/2010-editions/express。学生朋友们可以访问DreamSpark网站获取相应的开发工具。DreamSpark是由微软主办的一个项目，旨在帮助学生免费获取和体验微软正版开发工具。
- Microsoft .NET Framework 4.0，它会随Visual Studio 2010一起安装，或是随Visual Studio 2012一起安装的Microsoft .NET Framework 4.5。

## 2.2　配置开发环境

安装完上节提到的任一版本的Visual Studio后，就可以安装从微软Kinect for Windows网站下载的SDK了，下载地址是http://www.microsoft.com/en-us/kinectforwindows/。Kinect for Windows SDK由以下几部分内容组成：

- 在计算机上使用的Kinect驱动，支持Windows 7和Windows 8操作系统；
- API和设备接口；
- MSDN上为开发者提供的技术文档，使得开发更加便捷。

Kinect for Windows SDK的安装非常简单，这里只进行简要说明。在“最终用户许可协议”窗口中，仔细阅读许可协议，然后勾选“我同意许可条款和条件”复选框，如图2-1所示。

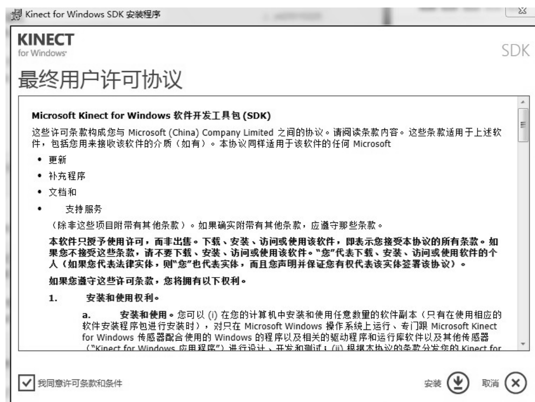



图2-1　安装许可协议

接着单击“安装”按钮，开始Kinect for Windows SDK的安装。安装程序会自动识别操作系统类型并选择安装相应的SDK，如图2-2所示。




图2-2　安装过程

安装结束以后，就会看到“安装已完成”界面，如图2-3所示，这说明Kinect for Windows SDK已经安装成功了。还会看到下载开发者工具包的提示，第7章将介绍开发者工具包的安装。




图2-3　SDK安装成功

此时，将Kinect for Windows设备连接到电脑上，系统会自动安装设备驱动，安装结果如图2-4所示。这时Kinect for Windows设备的绿色指示灯开始闪烁，表示驱动安装成功。




图2-4　驱动安装成功

## 2.3 Kinect for Windows SDK 技术架构

Kinect for Windows SDK提供了非常尖端和复杂的软件库及工具，帮助开发者充分利用基于Kinect的自然输入、信息捕获以及对真实世界事件的反应等特性进行开发工作。Kinect传感器通过该软件库与应用程序进行交互，如图2-5所示。

其中，Kinect for Windows SDK中的设备驱动程序首先从硬件读取原始数据，包括图像数据、深度数据和音频数据，然后在NUI类库中进行计算，得到骨骼点位置、声源位置等信息。而Kinect应用则通过与NUI类库中的接口进行交互，来获取所需的数据。




图2-5　应用程序与软硬件之间的交互

NUI（Natural User Interface，自然用户界面）是指一类无形的用户界面。“自然”一词是相对图形用户界面（GUI）而言的，GUI要求用户必须先学习软件开发者预先设置好的操作，而NUI则只需要人们以最自然的交流方式（如语言和文字）与机器互动。直观地说，使用NUI的计算机不需要键盘或鼠标。

NUI API是Kinect for Windows API的核心，负责获取Kinect设备捕获到的音频流以及彩色和深度图像流，并且控制Kinect设备，支持基本的图像和设备管理特性。NUI API的主要功能包括：

- 访问连接到电脑上的Kinect设备；
- 通过Kinect图像传感器获取彩色图像和深度数据；
- 传送处理过的彩色图像和深度数据；
- 利用彩色图像和深度数据来进行骨骼跟踪；
- 通过麦克风组获取音频数据。

PC上的Kinect驱动程序支持在一台电脑上使用多个Kinect设备。在NUI API中，包含计算Kinect传感器数量的函数，用户可以自己决定连接到电脑的Kinect设备数量，并且得到特定Kinect的ID，每个Kinect设备都可以独立打开或设置。

## 2.4　小结

本章主要介绍了Kinect for Windows开发环境的配置方法，可以看出配置过程十分简单。开发环境已经配置完成，从下一章开始，我们就要逐步学习Kinect开发的相关知识了，每一部分的知识都结合了实例程序，读者可以边学边做，快速入门。

*Part 2*

第二部分

# Kinect 开发篇

在开发 Kinect 应用之前，首先需要了解 Kinect for Windows SDK 的具体功能：能提供什么类型的数据，数据的组织方式是怎样的，使用什么方法获取数据，数据的规格是什么，等等。只有解决了这些基础的问题，才能够通过合适的方法来开发自己的 Kinect 应用。

因此，本书的第二部分将会围绕这些问题，对 Kinect for Windows SDK 中的 API 以及通过 API 获取的相关数据进行详细的介绍。同时，每一章都配有相应的实例程序来帮助大家更好地理解调用 API 的方法，并对获取到的各类数据格式有一个最直观的认识。

# 第3章

# Kinect彩色和红外图像数据的处理

前面介绍过，Kinect总共有3个摄像头，中间的一个是RGB摄像头，彩色图像就是通过该摄像头获取的。跟普通的摄像头一样，彩色图像数据流以一系列静态图像的形式传送给应用程序。两边的深度传感器会产生并接收随机分布的红外光线，用以确定物体距离Kinect传感器的距离并计算出深度图像，Kinect for Windows SDK提供了获取彩色图像和红外图像的API，下面将做详细介绍。

## 3.1　彩色图像的格式

从Kinect获取的彩色图像有两种质量——普通质量和高质量，而图像质量决定了数据从Kinect传输到PC的速度。下面简要介绍这两种图像质量。

- **普通质量**：彩色图像数据在传递给应用控制台之前会在传感器端进行压缩，接着在控制台解压数据，然后将数据传递到应用上。图像压缩使得返回的彩色数据的帧率高达30f/s，但是会降低图像质量。
- **高质量**：彩色图像数据在传感器端不会进行压缩，它将获取的原始数据直接传递给控制台。由于图像没有压缩，那么每帧就要传递更多的数据，因此最大帧率不会超过15f/s。此外，没有压缩的数据也需要系统分配更大的缓冲空间。

Kinect传感器通过USB连接到PC，该连接提供一个给定的带宽值。根据用户对图像质量的选择，可以调整使用的带宽值。如果选择高质量的图像，每帧会发送更多的数据，但更新比较慢；而如果选择普通质量的图像，更新较快，但会降低图像质量。

彩色数据可以选用两种色彩格式，这两种格式决定了返回应用的图像数据是以RGB形式还是以YUV形式编码。

- RGB格式在sRGB色彩空间提供32位线性X8R8G8B8格式的彩色位图。
- YUV格式提供16位伽马校正的线性UYVY格式的彩色位图，YUV色彩空间的伽马校正等价于RGB色彩空间的sRGB伽马校正。由于YUV流中每个像素只有16位，因此用这种格式保存位图数据时占用的存储空间较少，调用`NuiImageStreamOpen`函数时只需分配较小的缓存。

在实际编程中，可以根据应用程序实现的需要选择合适的图像数据格式。而在Kinect for Windows SDK的API中，彩色图像类型用枚举类型`ColorImageFormat`表示，可枚举的值如表3-1所示。

表3-1　**ColorImageFormat**类型的枚举值

| 成员名称 | 描　述 |
| --- | --- |
| `InfraredResolution640x480Fps30` | 红外数据，分辨率为640×480，帧率为15f/s |
| `RawBayerResolution1280x960Fps12` | 拜尔原始数据，分辨率为1280×960，帧率为12f/s |
| `RawBayerResolution640x480Fps30` | 拜尔原始数据，分辨率为640×480，帧率为30f/s |
| `RawYuvResolution640x480Fps15` | 数据类型为Raw YUV，分辨率为640×480，帧率为15f/s |
| `RgbResolution1280x960Fps12` | 数据类型为RGB，分辨率为1280×960，帧率为12f/s |
| `RgbResolution640x480Fps30` | 数据类型为RGB，分辨率为640×480，帧率为30f/s |
| `YuvResolution640x480Fps15` | 数据类型为 YUV，分辨率为640×480，帧率为15f/s |
| `Undefined` | 未定义的格式 |



## 3.2　红外数据流

介绍红外数据之前，首先简要介绍一下Kinect深度摄像机的工作原理。如1.3.1节所述，Kinect左右两侧的传感器分别负责发射和接收红外线：Kinect首先通过左侧的红外线发射器向环境中发射红外线，这束红外线由于具有高度随机性，其在空间中任意两个不同位置所反射形成的光斑都不相同，对环境形成立体的“光编码”；再通过右侧的红外线接收器来采集Kinect视野中的红外线图像；最终，利用这副红外图像和Kinect的原始参数进行一系列复杂的计算，就可以得到视野中的三维深度信息了，即下一章会详细介绍的深度数据。

红外图像数据需要作为一种彩色图像格式从SDK中获取，对应的图像格式为表3-1中的`InfraredResolution640x480Fps30`。因此，其处理方式跟彩色图像大体相同，而且需要特别注意的一点是，红外图像无法和彩色图像同时获取。

## 3.3　实例 1——调用 API 获取彩色图像数据和红外图像，并实现静态图像的抓取

本节主要讲解如何调用Kinect for Windows SDK相应的API获取彩色图像和红外图像数据，具体操作步骤如下。

(1) 创建WPF工程。打开Visual Studio 2010，新建一个WPF工程，将其命名为KinectColorViewer，如图3-1所示。

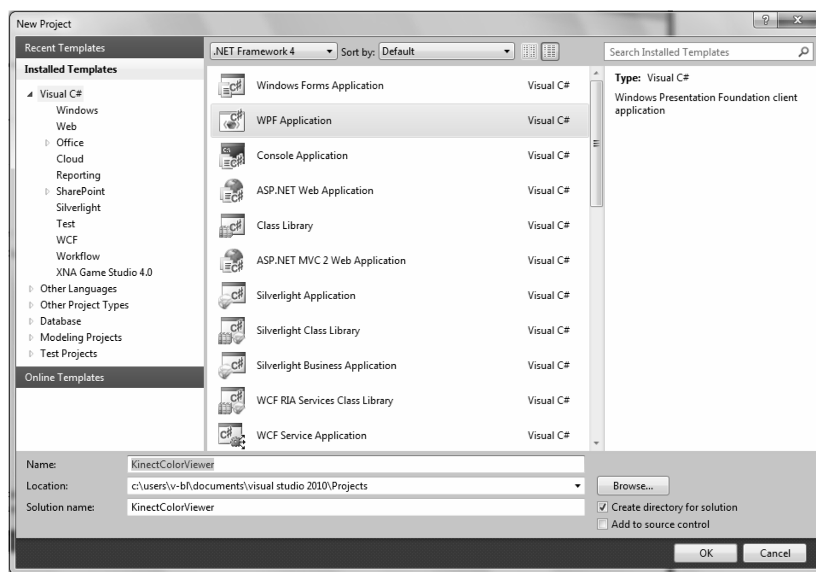



图3-1　新建WPF工程

(2) 添加Kinect for Windows SDK程序集的引用。在“Solution Explorer”一栏中，右击“References”，在弹出的快捷菜单中选择“Add Reference”菜单项，此时将打开“Add Reference”对话框，从中找到Microsoft.Kinect，并添加该引用，如图3-2和图3-3所示。

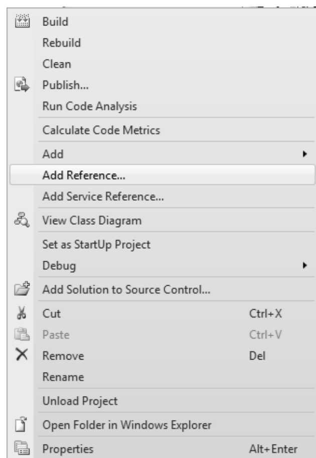



图3-2　添加引用选项

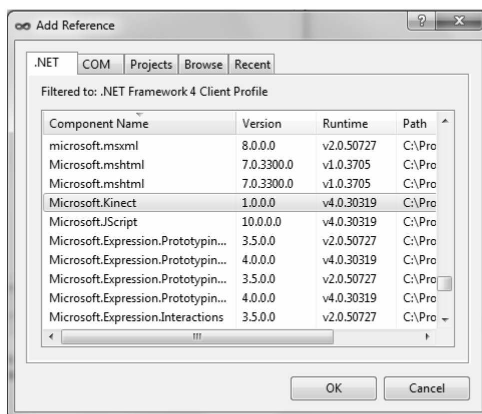



图3-3　Microsoft.Kinect库

(3) 添加控件。双击打开MainWindow.xaml文件，在设计器中添加一个Image控件，用于显示获取到的彩色图像。将这个Image控件命名为`ColorImage`，对应的MainWindow.xaml中的代码如下所示：

```
<Grid>
<Image Height="311" HorizontalAlignment="Left"
    Name="ColorImage" Stretch="Fill"  VerticalAlignment="Top" Width="503" />
</Grid>
```

(4) 引用命名空间。打开MainWindow.xaml.cs文件，在文件头添加对Kinect对象的引用，如下所示：

```
using Microsoft.Kinect;
```

(5) 添加窗口加载和关闭函数。打开MainWindow.xaml的设计器，在“Properties”窗口中选中“Events”选项卡，找到“Loaded”和“Closed”事件，分别双击即可添加相应的事件处理函数，如图3-4所示。



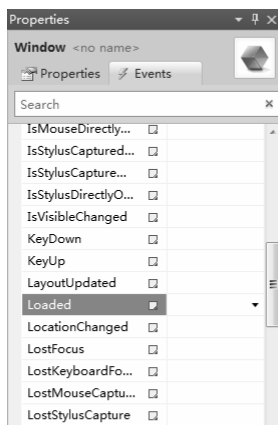



图3-4　Loaded事件添加

(6) 定义KinectSensor对象。KinectSensor类提供了一些接口，用于管理Kinect设备的开关以及所有数据的获取等，详细内容可以查阅Kinect for Windows SDK附带的技术文档。在MainWindow.xaml.cs文件的MainWindow类中，声明如下所示的两个变量，其中KinectSensor对象代表一个单独的Kinect设备，byte数组用来存放获取到的图像数据。

```
KinectSensor kinectSensor;
private byte[] pixelData;
```

(7) 在Loaded事件的处理函数中添加KinectSensor对象的初始化代码，如下所示：

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    kinectSensor = (from sensor in KinectSensor.KinectSensors
                    where sensor.Status == KinectStatus.Connected
                    select sensor).FirstOrDefault();
    kinectSensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
    kinectSensor.Start();
    kinectSensor.ColorFrameReady+=kinectSensor_ColorFrameReady;
}
```

由于Kinect for Windows SDK支持多个Kinect同时使用，这样通过KinectSensor类获取到的是一个KinectSensors数组。本实例使用LINQ语句将KinectSensors数组中KinectStatus为Connected的项筛选出来，并将得到的Kinect的列表中第一个或是默认的Kinect设备赋值给kinectSensor。

然后通过kinectSensor启用视频流，在ColorStream的Enable()方法中初始化视频数据的格式、分辨率以及帧率。

接着调用kinectSensor的Start()方法启动设备，开始接收视频流。每当下一帧的数据准备好时，ColorFrameReady事件会通知应用程序添加一个kinectSensor_ColorFrameReady事件处理函数，来获取该事件返回的通知。

使用完Kinect后必须将其关闭，可以通过在Closed事件中调用kinectSensor的Stop()方法来实现，具体代码如下所示：

```
private void Window_Closed(object sender, EventArgs e)
{
    kinectSensor.Stop();
}
```

(8) 接受视频数据。在kinectSensor_ColorFrameReady事件处理函数中获取视频数据，并将获取到的数据显示出来。该事件处理函数的定义如下：

```
private void kinectSensor_ColorFrameReady(object sender,
    ColorImageFrameReadyEventArgs e)
{
    using (ColorImageFrame imageFrame = e.OpenColorImageFrame())
    {
        if (imageFrame != null)
        {
            this.pixelData = new byte[imageFrame.PixelDataLength];
            imageFrame.CopyPixelDataTo(this.pixelData);
            this.ColorImage.Source = BitmapSource.Create(imageFrame.Width,
                                     imageFrame.Height, 96, 96,
                                     PixelFormats.Bgr32, null, pixelData,
                                     imageFrame.Width * imageFrame.BytesPerPixel);
        }
    }
}
```

ColorFrameReady事件会给事件处理函数传递一个ColorImageFrameReadyEventArgs参数e，通过调用其中的OpenColorImageFrame()方法获取从Kinect设备返回的下一帧数据。如果获取到了这一数据，就调用CopyPixelDataTo()方法将数据复制到已分配好空间的byte数组，该数组的大小由ColorImageFrame的PixelDataLength确定。

最后利用获取到的数据创建一个BitmapSource，并将其赋值给最初添加的Image控件ColorImage的Source属性。这样就可以不断地将获取到的下一帧图像数据刷新显示到界面上。

(9) 运行程序。按Ctrl+F5组合键运行程序，显示结果如图3-5所示。

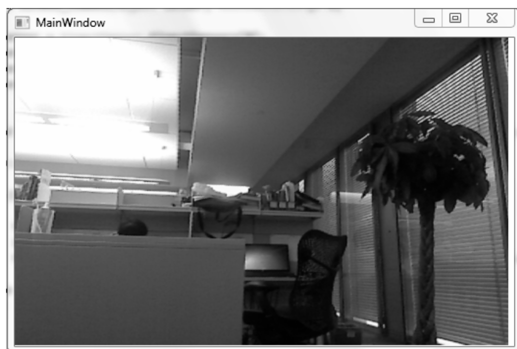



图3-5　获取彩色图像运行结果

(10) 红外图像实际上是彩色图像的一种特殊格式，因此红外图像的获取方式跟彩色图像一样，只需在上述代码中进行两处修改即可。将

```
kinectSensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
```

这条语句中的图像格式改为

```
kinectSensor.ColorStream.Enable(ColorImageFormat.InfraredResolution640x480Fps30);
```

这是因为`ColorImageFormat.InfraredResolution640x480Fps30`是红外图像的格式类型。同时，由于红外图像为16位的灰度图像，因此需要将显示图像语句

```
this.ColorImage.Source = BitmapSource.Create(imageFrame.Width,
    imageFrame.Height, 96, 96, PixelFormats. Bgr32, null, pixelData,
    imageFrame.Width * imageFrame.BytesPerPixel);
```

改为

```
this.ColorImage.Source = BitmapSource.Create(imageFrame.Width,
    imageFrame.Height, 96, 96, PixelFormats. Gray16, null, pixelData,
    imageFrame.Width *  imageFrame.BytesPerPixel);
```

重新编译，运行程序，即可得到红外图像，如图3-6所示。




图3-6　获取红外图像运行结果

## 3.4　小结

彩色数据是Kinect开发中一种最基本的数据类型，它和普通摄像头采集到的数据一样，可以用于开发基于图像的应用程序。另外，我们还可以结合后面讲到的深度数据和骨骼追踪数据开发出更多有趣的体感交互应用程序。

而红外图像数据作为深度数据和骨骼追踪数据的前身，由于其数据类型过于原始而很少在应用开发中直接使用。不过相信在不久的将来，一定会涌现出更多利用红外图像数据开发出来的优秀应用。

# 第4章 Kinect深度数据的处理



Kinect两侧的传感器负责获取深度数据，而深度数据是指Kinect视野范围内的物体到Kinect的三维空间距离。本章将首先对深度数据的结构进行讲解，接着结合一个实例程序介绍如何获取并处理深度数据。

## 4.1 深度数据的结构

深度数据流提供了一种结构，该结构中每个像素的高13位表示在深度传感器的视野范围内离特定坐标物体最近的距离（单位：mm）。有两种深度数据流可以使用：

- 帧的大小为320×240像素；
- 帧的大小为80×60像素。

在Kinect for Windows SDK中，Kinect通过处理深度数据来识别传感器组前的两个人体图像，然后创建玩家分段图。该图是一张位图，其像素值与视野内距离摄像头最近的玩家索引对应。

尽管玩家分段数据是隔离的逻辑流，但实际上深度数据和玩家分段数据被合并到了一个独立的结构中：

- 每个像素的高13位表示从深度传感器到最近的物体的距离，单位为mm；
- 每个像素的低3位表示在像素的*xy*坐标系上追踪到的可见的玩家索引，这3位可看做整型值。

玩家索引值为0，表示在相应位置没有找到玩家，索引值为1和2表示检测到的玩家编号，依次类推。目前深度数据最多可以检测到6名玩家。玩家分段数据常用来隔离特定的玩家，或是从原始的彩色深度图像中分离出感兴趣的区域。

## 4.2 实例2——调用API获取深度数据，并对不同深度值着以不同颜色

本实例程序主要实现从Kinect获取视野内物体的深度值，并且根据物体距离Kinect的远近着以不同的颜色。前面的部分实现细节同上一章的实例程序一样，这里简要略过。

(1) 配置开发环境。同第3章的实例一样，首先新建一个WPF工程，命名为KinectDepthViewer，接着添加Kinect for Windows SDK程序集的引用。

(2) 同样添加一个Image控件，命名为`DepthImage`，用于显示获取到的深度图像。

```
<Grid>
    <Image Height="311"
           HorizontalAlignment="Left"
           Name="DepthImage"
           Stretch="Fill"
           VerticalAlignment="Top"
           Width="503" />
</Grid>
```

(3) 在MainWindow.xaml.cs文件中引入NUI的命名空间。

```
using Microsoft.Kinect;
```

(4) 在MainWindow.xaml.cs文件的MainWindow类中，声明如下所示的两个变量，其中KinectSensor对象代表一个单独的Kinect设备；short数组用来存放获取到的深度图像数据。前面讲到，深度图像数据中每个像素都是用16位表示的，因此使用short数组。

```
KinectSensor kinectSensor;
private short[] pixelData;
```

(5) 添加窗口加载和关闭函数，具体代码如下：

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    kinectSensor = (from sensor in KinectSensor.KinectSensors
                    where sensor.Status == KinectStatus.Connected
                    select sensor).FirstOrDefault();
    kinectSensor.DepthStream.Enable(DepthImageFormat.Resolution320x240Fps30);
    kinectSensor.Start();
    kinectSensor.DepthFrameReady += new
        EventHandler<DepthImageFrameReadyEventArgs>(kinectSensor_DepthFrameReady);
}

private void Window_Closed(object sender, EventArgs e)
{
    kinectSensor.Stop();
}
```

与上一章的实例类似，首先使用LINQ语句筛选连接到电脑的Kinect设备，并赋值给kinectSensor。然后调用DepthStream的Enable()函数启用深度图像数据，其参数为Resolution320x240Fps30，含义是设置深度图像的分辨率为320×240，帧率为30f/s。

接下来，调用Start()函数启动Kinect数据流。每当下一帧的深度图像数据准备好时，DepthFrameReady事件会通知应用程序添加一个kinectSensor_DepthFrameReady事件处理函数，以获取该事件返回的通知，参数类型为DepthImageFrameReadyEventArgs。

(6) 获取深度图像数据。在kinectSensor_DepthFrameReady事件的处理函数中获取深度图像数据，并将其显示在界面上。kinectSensor_DepthFrameReady事件处理函数的定义如下所示。

```
private void kinectSensor_DepthFrameReady(object sender,
    DepthImageFrameReadyEventArgs e)
{
```

```
        using (DepthImageFrame depthImageFrame = e.OpenDepthImageFrame())
        {
            if (depthImageFrame != null)
            {
                pixelData = new short[depthImageFrame.PixelDataLength];
                depthImageFrame.CopyPixelDataTo(pixelData);
                this.DepthImage.Source = BitmapSource.Create(depthImageFrame.Width,
                    depthImageFrame.Height, 96, 96, PixelFormats.Gray16, null, pixelData,
                    depthImageFrame.Width * depthImageFrame.BytesPerPixel);
            }
        }
    }
```

可以看到，上述代码调用OpenDepthImageFrame()函数来获取深度图像数据，深度图像数据类型定义为DepthImageFrame，关于该类型的详细介绍可以参阅SDK附带的技术文档。depthImageFrame不为空代表获取到了深度图像数据，这时可调用DepthImageFrame的CopyPixelDataTo()方法，将从Kinect设备获取到的深度图像数据复制到short数组。当然，在这之前要给pixelData分配相应的空间，大小由depthImageFrame的PixelDataLength属性决定。



最后利用获取到的数据创建一个BitmapSource，并将其赋值给最初添加的Image控件ColorImage的Source属性。这样就可以不断地将获取到的下一帧的深度图像数据刷新显示到界面上。需要注意，与上一章实例不同的是，像素格式PixelFormats的值为Gray16，因为原始的深度图像数据是16位灰度图像。

(7) 运行程序，得到的显示结果如图4-1所示。

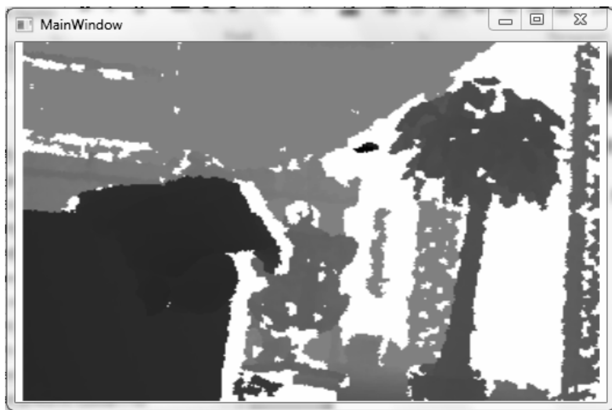



图4-1　灰度图像获取结果

图中显示的是原始的灰度图像。为了使读者对深度图像的数据结构有更深入的理解，接下来将介绍如何获取每个像素点的深度值，并针对不同的深度值着以不同的颜色。

我们首先来分析深度数据的结构特点。上面已经提到，调用DepthImageFrame的CopyPixelDataTo()方法，可以将从Kinect获取到的深度数据复制到short类型的pixelData

数组，该数组包含了每个像素的深度信息和玩家索引信息。由于每个像素都用16位表示（低3位表示玩家索引，高13位表示深度值），因此我们可以通过移位操作得到相应的值。对每个像素的值只取低3位便可得到玩家索引值，取值范围为0 ~ 7，0代表没有玩家，1代表玩家1，2代表玩家2，依次类推；将该像素的值右移3位即可得到深度值，理论上该值的范围为0 ~ 8192 毫米。下列代码表示获取第*i*个像素点的玩家索引和实际深度值。其中，`PlayerIndexBitmask`和`PlayerIndexBitmaskWidth`是`DepthImageFrame`的两个常量属性，分别为7和3：

```
int player = depthFrame[i] & DepthImageFrame.PlayerIndexBitmask;
int realDepth = depthFrame[i] >> DepthImageFrame.PlayerIndexBitmaskWidth;
```

在了解了深度数据的结构之后，接下来就需要写一个函数`ConvertDepthFrame()`，将得到的原始灰度深度数据转化为彩色数据。在`MainWindow`类中定义一个`byte`数组`depthFrame32`来存储彩色图像数据，此外还需要定义3个常量，用来索引每个像素的RGB值，具体代码如下：

```
byte[] depthFrame32;
private const int RedIndex = 2;
private const int GreenIndex = 1;
private const int BlueIndex = 0;
```

`ConvertDepthFrame()`函数的实现代码如下：

```
private byte[] ConvertDepthFrame(short[] depthFrame, DepthImageStreamdepthStream)
{
    int tooNearDepth = depthStream.TooNearDepth;
    int tooFarDepth = depthStream.TooFarDepth;
    int unknownDepth = depthStream.UnknownDepth;

    for (int i = 0, j = 0; i<depthFrame.Length&& j < this.depthFrame32.Length;
        i++, j += 4)
    {
        int player = depthFrame[i] & DepthImageFrame.PlayerIndexBitmask;
        int realDepth = depthFrame[i] >> DepthImageFrame.PlayerIndexBitmaskWidth;
        //根据深度值的不同着以不同的颜色
        if (player == 0 && realDepth == 0)
        {
            //白色
            this.depthFrame32[j + RedIndex] = 255;
            this.depthFrame32[j + GreenIndex] = 255;
            this.depthFrame32[j + BlueIndex] = 255;
        }
        else if (player == 0 &&realDepth>0 &&realDepth<= tooNearDepth)
        {
            //青色
            this.depthFrame32[j + RedIndex] = 0;
            this.depthFrame32[j + GreenIndex] = 255;
            this.depthFrame32[j + BlueIndex] = 255;
        }
        else if (player == 0 &&realDepth>tooNearDepth&&realDepth<tooFarDepth)
        {
            //紫色
            this.depthFrame32[j + RedIndex] = 160;
            this.depthFrame32[j + GreenIndex] = 32;
            this.depthFrame32[j + BlueIndex] = 240;
```

```
            }
            else if (player == 0 &&realDepth>= tooFarDepth)
            {
                //灰色
                this.depthFrame32[j + RedIndex] = 192;
                this.depthFrame32[j + GreenIndex] = 192;
                this.depthFrame32[j + BlueIndex] = 192;
            }
            else if (player == 0 &&realDepth == unknownDepth)
            {
                //黄色
                this.depthFrame32[j + RedIndex] = 255;
                this.depthFrame32[j + GreenIndex] = 255;
                this.depthFrame32[j + BlueIndex] = 0;
            }
            else if(player>0)
            {
                switch (player)
                {
                case 1: //红色
                    this.depthFrame32[j + RedIndex] = 255;
                    this.depthFrame32[j + GreenIndex] = 0;
                    this.depthFrame32[j + BlueIndex] = 0;
                    break;
                case 2: //绿色
                    this.depthFrame32[j + RedIndex] = 0;
                    this.depthFrame32[j + GreenIndex] = 255;
                    this.depthFrame32[j + BlueIndex] = 0;
                    break;
                default: //蓝色
                    this.depthFrame32[j + RedIndex] = 0;
                    this.depthFrame32[j + GreenIndex] = 0;
                    this.depthFrame32[j + BlueIndex] = 255;
                    break;
                }
            }
        }
        return this.depthFrame32;
    }
```

可以看到，转化函数需要传递两个参数，第一个参数不用多说，肯定是指存储获取到的原始深度数据的数组；第二个参数为深度数据流DepthImageStream，是因为要用到该类型的几个属性值。

为了给不同的深度值着以不同的颜色，这里直接将DepthImageStream的3个属性值作为临界点，该属性值的含义如表4-1所示。

**表4-1　DepthImageStream属性**

| 属　性　名 | 含　　义 |
| --- | --- |
| TooNearDepth | 距离Kinect太近的最大深度值，单位：mm |
| TooFarDepth | 距离Kinect太远的最小深度值，单位：mm |
| UnknownDepth | 获取不到深度值时返回的值 |

在kinectSensor_DepthFrameReady事件的处理函数中添加对ConvertDepthFrame的调用，更新后的kinectSensor_DepthFrameReady如下所示：

```
private void kinectSensor_DepthFrameReady(object sender,
    DepthImageFrameReadyEventArgs e)
{
    using (DepthImageFrame depthImageFrame = e.OpenDepthImageFrame())
    {
        if (depthImageFrame != null)
        {
            pixelData = new short[depthImageFrame.PixelDataLength];
            depthImageFrame.CopyPixelDataTo(pixelData);

            this.depthFrame32 = new byte[depthImageFrame.Width *
                depthImageFrame.Height * 4];
            this.depthFrame32 = ConvertDepthFrame(pixelData, ((KinectSensor)
                sender).DepthStream);
            this.DepthImage.Source = BitmapSource.Create(depthImageFrame.Width,
                depthImageFrame.Height, 96, 96, PixelFormats.Bgr32, null,
                depthFrame32, depthImageFrame.Width * 4);
        }
    }
}
```

此时再运行程序，显示结果如图4-2所示。




图4-2　彩色图像获取结果

## 4.3　小结

由于深度数据中包含了Kinect视野范围内的物体到Kinect的实际距离，因此根据深度值的不同，应用程序可以很容易地将前景物体和背景分离开来，并且能够从任意一种深度数据流中处理数据，以支持多样的常规特性，例如跟踪用户的运动，识别物体背景以便在应用播放时忽略掉背景。这一特性在第9章中得到了很好的应用。

第 5 章

# Kinect骨骼追踪数据的处理方法

5



骨骼追踪技术是Kinect的核心技术，它可以准确标定人体的20个关键点，并能对这20个点的位置进行实时追踪。利用这项技术，可以开发出各种基于体感人机交互的有趣应用。

## 5.1　骨骼追踪数据的结构

目前，Kinect for Windows SDK中的骨骼API可以提供位于Kinect前方至多两个人的位置信息，包括详细的姿势和骨骼点的三维坐标信息。另外，Kinect for Windows SDK最多可以支持20个骨骼点。数据对象类型以骨骼帧的形式提供，每一帧最多可以保存20个点，如图5-1所示。

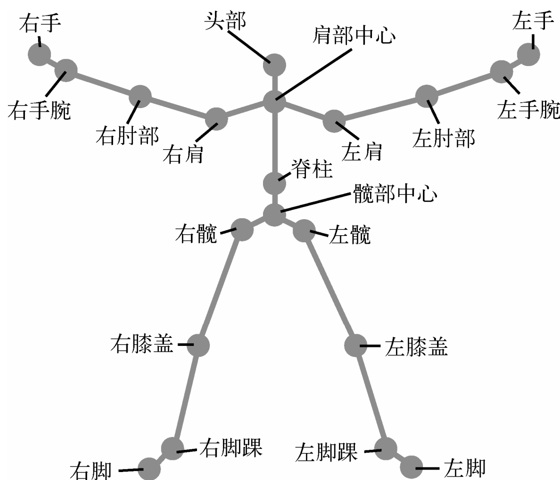



图5-1　20个骨骼点示意图

在SDK中每个骨骼点都是用`Joint`类型来表示的，每一帧的20个骨骼点组成基于`Joint`类型的集合。此类型包含3个属性，具体内容如下所示。

- JointType：骨骼点的类型，这是一种枚举类型，列举出了20个骨骼点的特定名称，比如“HAND_LEFT”表示该骨骼点是左手节点。
- Position：SkeletonPoint类型表示骨骼点的位置信息。SkeletonPoint是一个结构体，包含X、Y、Z三个数据成员，用以存储骨骼点的三维坐标。
- TrackingState：JointTrackingState类型也是一种枚举类型，表示该骨骼点的追踪状态。其中，Tracked表示正确捕捉到该骨骼点，NotTracked表示没有捕捉到骨骼点，Inferred表示状态不确定。

## 5.2　半身模式

如果应用程序只需要捕捉上半身的姿势动作，就可以采用Kinect for Windows SDK提供的半身模式（Seated Mode）。在半身模式下，系统只捕捉人体上半身10个骨骼点的信息，而忽略下半身另外10个骨骼点的位置信息，这样就解决了用户坐在椅子上时无法被Kinect识别的问题，即使下半身骨骼点的数据不稳定或是不存在也不会对上半身的骨骼数据造成影响。而且当用户距离Kinect设备只有0.4m时，应用程序仍能正常地进行骨骼追踪，这就大幅提高了骨骼追踪的性能。

半身模式定义在枚举类型SkeletonTrackingMode中，该类型包含两个枚举值：Default和Seated。前者为默认的骨骼追踪模式，会正常捕捉20个骨骼点；后者为半身模式，选择该值则只捕捉上半身的10个骨骼点。

开发者可以通过改变SkeletonStream对象的TrackingMode属性来设置骨骼追踪的模式，代码如下：

```
kinectSensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Seated;
```

## 5.3　骨骼追踪数据的获取方式

应用程序获取下一帧骨骼数据的方式同获取彩色图像和深度图像数据的方式一样，都是通过调用回调函数并传递一个缓存实现的，获取骨骼数据调用的是OpenSkeletonFrame()函数。如果最新的骨骼数据已经准备好了，那么系统就会将其复制到缓存中；但如果应用程序发出请求时，新的骨骼数据还未准备好，此时可以选择等待下一个骨骼数据直至其准备完毕，或者立即返回稍后再发送请求。对于NUI骨骼API而言，相同的骨骼数据只会提供一次。

NUI骨骼API提供了两种应用模型，分别是轮询模型和时间模型，简要介绍如下。

- 轮询模型是读取骨骼事件最简单的方式，通过调用SkeletonStream类的OpenNextFrame()函数即可实现。OpenNextFrame()函数的声明如下所示。

  ```
  public SkeletonFrame OpenNextFrame (
       int millisecondsWait
  )
  ```

  可以传递参数指定等待下一帧骨骼数据的时间。当新的数据准备好或是超出等待时间时，OpenNextFrame()函数才会返回。

- 时间模型以事件驱动的方式获取骨骼数据，更加灵活、准确。应用程序传递一个事件处理函数给SkeletonFrameReady事件，该事件定义在KinectSensor类中。当下一帧的骨骼数据准备好时，会立即调用该事件回调函数。因此Kinect应用应该通过调用OpenSkeletonFrame()函数来实时获取骨骼数据。

## 5.4 实例 3——调用 API 获取骨骼数据并实时绘制

本实例程序将实现获取骨骼数据，然后将骨骼点的坐标作为Ellipse控件的20个位置坐标，同时用线段将相应的点连接起来，最后将绘制出的骨架映射到彩色图像上。读者可以在实例1的基础上开始本实例，具体操作步骤如下所示。

(1) 在Window_Loaded()函数中添加下列骨骼数据流的启动函数，并添加 kinectSensor_SkeletonFrameReady事件处理函数相应的SkeletonFrameReady事件。

```
kinectSensor.SkeletonStream.Enable();
kinectSensor.SkeletonFrameReady += new
    EventHandler<SkeletonFrameReadyEventArgs>(kinectSensor_SkeletonFrameReady);
```

(2) 准备WPF界面。通过以下代码在界面上添加20个小圆点，分别跟踪由Kinect for Windows SDK获取到的人体的20个关键点，并将这20个点标记为不同的颜色。

```
<Canvas Name="SkeletonCanvas" Visibility="Visible">
    <Ellipse Canvas.Left="0" Canvas.Top="0" Height="10" Name="headPoint"
        Width="10" Fill="Red" />
    <Ellipse Canvas.Left="10" Canvas.Top="0" Height="10"
        Name="shouldercenterPoint" Width="10" Fill="Blue" />
    <Ellipse Canvas.Left="20" Canvas.Top="0" Height="10"
        Name="shoulderrightPoint" Width="10" Fill="Orange" />
    ……省略中间的Ellipse定义
    <Image Canvas.Left="303" Canvas.Top="161" Height="150" Name="image1"
        Stretch="Fill" Width="200" />
</Canvas>
```

此时，设计窗口如图5-2所示。




图5-2 WPF设计界面

(3) 编写kinectSensor_SkeletonFrameReady()事件处理函数。正确连接Kinect后，当用户站在Kinect前并且Kinect能够正确识别人体时，将触发该事件处理函数，其代码如下：

```
private void kinectSensor_SkeletonFrameReady(object sender,
    SkeletonFrameReadyEventArgs e)
{
    using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
    {
        if (skeletonFrame != null)
        {
            skeletonData = new
                Skeleton[kinectSensor.SkeletonStream.FrameSkeletonArrayLength];
            skeletonFrame.CopySkeletonDataTo(this.skeletonData);
            Skeleton skeleton = (from s in skeletonData
                where s.TrackingState == SkeletonTrackingState.Tracked
                select s).FirstOrDefault();
            if (skeleton!=null)
            {
              SetAllPointPosition(skeleton);
            }
        }
    }
}
```

上述代码使用LINQ语句来获取TrackingState等于Tracked的骨骼数据。目前SDK最多可以追踪两幅骨骼。为了简化起见，本实例只对捕捉到的第一幅骨骼进行追踪和显示。

(4) 在Skeleton对象的Joints属性集合中保存了所有骨骼点的信息，每个骨骼点的信息都是一个Joint对象。为了得到特定的骨骼点，同样使用LINQ语句对Joint的JointType属性进行筛选，相关代码如下：

```
Joint headJoint = (from j in skeleton.Joints
                   where j.JointType == JointType.Head
                   select j).FirstOrDefault();
```

在本实例程序中，需要遍历每个骨骼点，并分别对其进行处理。这里使用foreach语句来实现，并根据JointType属性进行处理。在SetAllPointPosition()函数中可以看到具体的实现细节。

```
foreach (Joint joint in skeleton.Joints)
{
    Point jointPoint = GetDisplayPosition(joint);
    switch (joint.JointType)
    {
        case JointType.Head:
            SetPointPosition(headPoint, joint);
            headPolyline.Points.Add(jointPoint);
            break;
            ……
    }
}
```

(5) 前面提到，`Joint`的`Position`属性的`X`、`Y`、`Z`表示该骨骼点的三维位置，其中`X`和`Y`的范围都是−1 ~ 1，而`Z`是Kinect到识别物体的距离。

为了能更好地将这20个点显示出来，需要对`Position`的X值和Y值进行缩放，可以通过以下函数实现。

```
private Point GetDisplayPosition(Joint joint)
{
    var scaledJoint = joint.ScaleTo(640, 480);
    return new Point(scaledJoint.Position.X, scaledJoint.Position.Y);
}
```

上面语句中，`ScaleTo`函数的最后两个参数640和480分别代表原始数据`X`和`Y`的最大值，通过该语句可以将*x*坐标放大到0 ~ 640范围内的任意值，将*y*坐标放大到0 ~ 480范围内的任意值。该坐标是相对于应用程序窗口的左上角(0,0)而言的，窗口的宽和高分别是640和480，以保证彩色图像和骨骼绘制的结果相匹配。

其中，`ScaleTo()`函数是`Coding4Fun`的`Help`类中的方法。`Coding4Fun`是一个Kinect开发辅助类库，本书将在第8章中对其进行详细的介绍。读者可以从http://c4fkinect.codeplex.com/下载该类库，并通过“Add Reference”菜单项将Coding4Fun.Kinect.Wpf.dll添加到项目中。



(6) 编写一个函数，将每个骨骼点转换后的（*x*, *y*）坐标值分别映射到相应的Ellipse控件的`Left`和`Top`属性上，其代码如下：

```
private void SetPointPosition(FrameworkElement ellipse, Joint joint)
{
    var scaledJoint = joint.ScaleTo(640, 480);
    Canvas.SetLeft(ellipse, scaledJoint.Position.X);
    Canvas.SetTop(ellipse, scaledJoint.Position.Y);
    SkeletonCanvas.Children.Add(ellipse);
}
```

使用`Polyline`类表示骨架线，显而易见，骨架由5条多段线组成，分别定义它们，并在遍历所有骨骼点时分类存储相应的点。详见`SetAllPointPosition()`函数，相关代码如下：

```
Polyline headPolyline = new Polyline();
Polyline handleftPolyline = new Polyline();
Polyline handrightPolyline = new Polyline();
Polyline footleftPolyline = new Polyline();
Polyline footrightPolyline = new Polyline();

private void SetAllPointPosition(Skeleton skeleton)
{
    SkeletonCanvas.Children.Clear();
    headPolyline.Points.Clear();
    handleftPolyline.Points.Clear();
    handrightPolyline.Points.Clear();
    footleftPolyline.Points.Clear();
    footrightPolyline.Points.Clear();

    foreach (Joint joint in skeleton.Joints)
    {
```

```
            Point jointPoint = GetDisplayPosition(joint);
            switch (joint.JointType)
            {
                case JointType.Head:
                    SetPointPosition(headPoint, joint);
                    headPolyline.Points.Add(jointPoint);
                    break;

                case JointType.ShoulderCenter:
                    SetPointPosition(shouldercenterPoint, joint);
                    headPolyline.Points.Add(jointPoint);
                    handleftPolyline.Points.Add(jointPoint);
                    handrightPolyline.Points.Add(jointPoint);
                    break;

                case JointType.ShoulderLeft:
                    SetPointPosition(shoulderleftPoint, joint);
                    handleftPolyline.Points.Add(jointPoint);
                    break;
                    ...
                case JointType.FootRight:
                    SetPointPosition(footrightPoint, joint);
                    footrightPolyline.Points.Add(jointPoint);
                    break;

                default:
                    ;
                    break;
            }
        }
        headPolyline.Stroke = new SolidColorBrush(Colors.Blue);
        headPolyline.StrokeThickness = 5;
        SkeletonCanvas.Children.Add(headPolyline);

        handleftPolyline.Stroke = new SolidColorBrush(Colors.Blue);
        handleftPolyline.StrokeThickness = 5;
        SkeletonCanvas.Children.Add(handleftPolyline);

        handrightPolyline.Stroke = new SolidColorBrush(Colors.Blue);
        handrightPolyline.StrokeThickness = 5;
        SkeletonCanvas.Children.Add(handrightPolyline);

        footleftPolyline.Stroke = new SolidColorBrush(Colors.Blue);
        footleftPolyline.StrokeThickness = 5;
        SkeletonCanvas.Children.Add(footleftPolyline);

        footrightPolyline.Stroke = new SolidColorBrush(Colors.Blue);
        footrightPolyline.StrokeThickness = 5;
        SkeletonCanvas.Children.Add(footrightPolyline);
    }
```

(7) 运行程序，显示结果如图5-3所示。

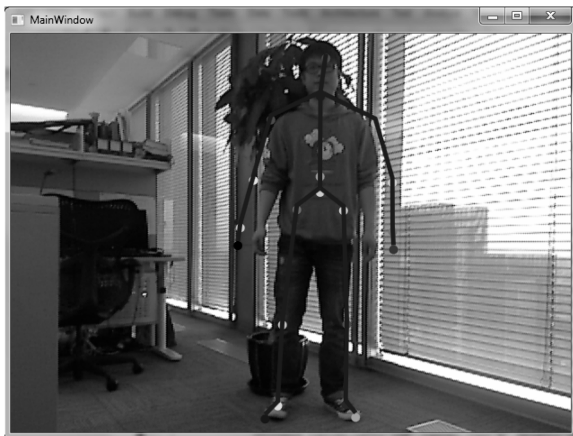



图5-3　全身骨骼点运行结果



(8) 若要使用半身模式，只需在初始化kinectSensor对象时添加以下语句即可。

```
kinectSensor.SkeletonStream.TrackingMode = SkeletonTrackingMode.Seated;
```

运行结果如图5-4所示。

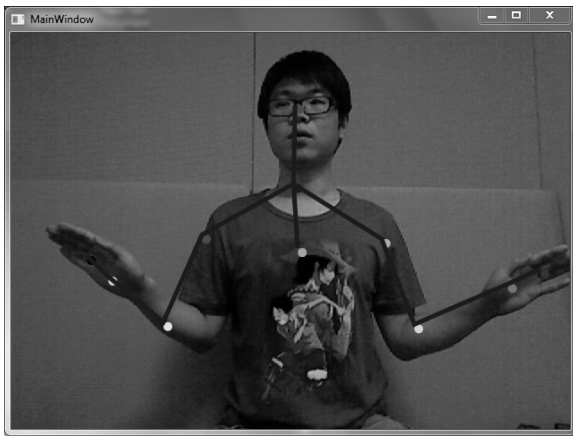



图5-4　半身模式运行结果

由于RGB图像数据与深度图像数据（骨骼数据）的空间坐标系是不同的，前者的原点是RGB摄像头，后者的原点是红外摄像头，因此本实例中使用获取的骨骼点坐标直接绘制在RGB图像上会有相应的误差。若要修正这些误差，可以调用Kinect for Windows SDK提供的映射函数，将骨骼点坐标映射到RGB图像坐标上。具体做法为将上面用的ScaleTo函数替换为MapSkeletonPointToColorPoint，使用方法如下所示：

```
ColorImagePoint colorImagePoint = kinectSensor.CoordinateMapper.MapSkeletonPointTo
ColorPoint(joint.Position, ColorImageFormat.RgbResolution640x480Fps30);
```

## 5.5　骨骼点旋转信息

除了跟踪骨骼点的位置，Kinect SDK还能计算出骨骼点的旋转信息。这是Kinect SDK 1.5版本新增的功能，利用此功能可以计算出人体骨骼在yaw轴的旋转情况，在此之前，仅通过骨骼点位置是无法实现此类计算的。根据相对参照系不同，旋转信息可以分为相对旋转信息和绝对旋转信息，这两种信息均包含了其旋转的矩阵参数和四元数参数。开发者可以使用这些数据方便地进行动作识别以及控制人形3D模型。

### 5.5.1　骨骼点旋转信息存储方式

在Kinect SDK中，骨骼点旋转信息定义为`BoneOrientation`类，包含以下数据成员。

- `StartJoint`：起始骨骼点；
- `EndJoint`：结束骨骼点；
- `HierarchicalRotation`：相对旋转信息；
- `AbsoluteRotation`：绝对旋转信息。

在学习相对旋转信息和绝对旋转信息的具体含义之前，我们首先要定义骨骼点坐标系。对Kinect跟踪到的20个骨骼点进行分层：将“髋部中心”作为初始骨骼点，相邻的骨骼点逐层向下延伸，如图5-5所示。

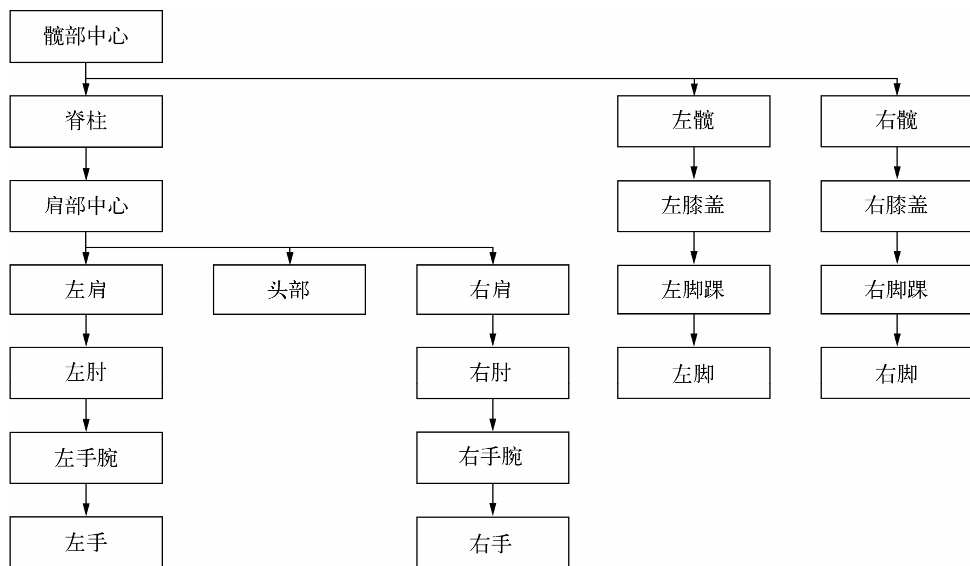



图5-5　骨骼点分层图

而骨骼点坐标系即为以该骨骼点为原点，以其上层骨骼点到它的直线方向为$y$轴正方向的坐标系，如图5-6所示。

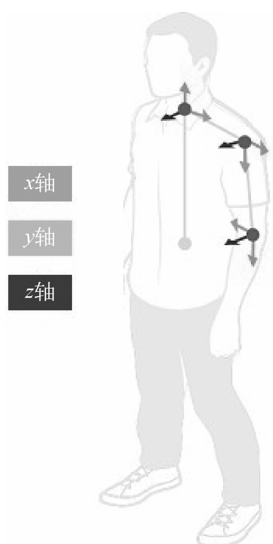



图5-6 相对旋转信息

其中，相对旋转信息就代表了一段骨骼中，起始骨骼点和结束骨骼点的两个坐标系之间的转移参数。相应地，绝对旋转信息代表了结束骨骼点坐标系和Kinect空间坐标系之间的转移参数，如图5-7所示。

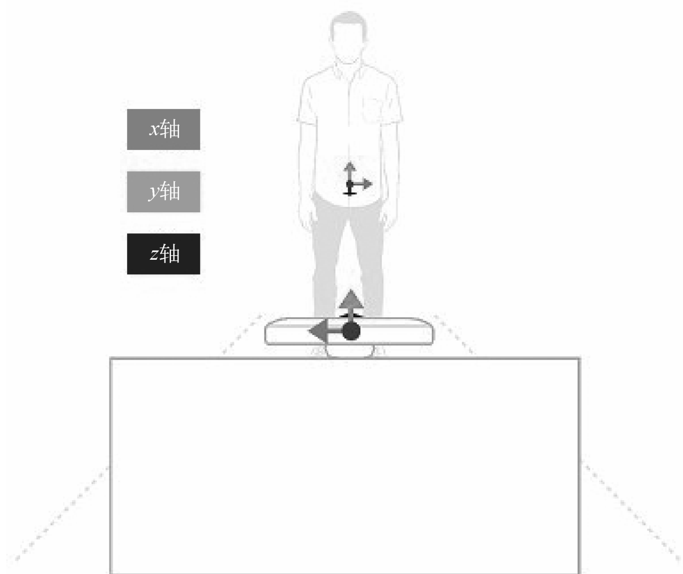



图5-7 绝对旋转信息

### 5.5.2　在骨骼数据回调函数中获取骨骼点旋转信息

由于骨骼点旋转信息包含在骨骼数据流中，因此需要在骨骼数据的回调函数中获取相应的数据。在获取了一帧SkeletonFrame中的SkeletonData之后，可以使用下列代码读取骨骼点旋转信息：

```
foreach (Skeleton skeleton in this.skeletonData)
{
    if (skeleton.TrackingState == SkeletonTrackingState.Tracked)
    {
        foreach (BoneOrientation orientation in skeleton.BoneOrientations)
        {
            BoneRotation hierarchical = orientation.HierarchicalRotation;
            BoneRotation absolute = orientation.AbsoluteRotation;
        }
    }
}
```

可以看出，读取骨骼点旋转信息的方法和读取骨骼数据类似，其中BoneRotation类型的数据记录了旋转信息的矩阵和四元数。

### 5.5.3　综述

虽然骨骼点旋转信息仅仅是依靠骨骼数据计算出来的，但是利用这一数据可以极其简便地完成人体姿态和动作的识别以及三维模型控制。不过需要注意的是，这里得到的旋转信息是由骨骼数据计算而来的原始数据，由于骨骼跟踪数据本身的抖动原因，旋转信息也会产生很大的噪声。因此，必须要先对其进行一定程度的降噪和平滑处理，才能在应用程序中使用。

## 5.6　实例 4——使用 Kinect 控制 PPT 播放

目前控制PPT切换的传统方式是使用鼠标、键盘或者手持遥控笔，通过按键实现幻灯片的换页。然而结合Kinect的体感交互技术，我们只需一个手势就可以完成PPT的换页，这是不是很炫呢？其实这个功能实现起来一点都不难，本实例将使用前面介绍的骨骼数据来实现手势控制PPT播放的简单功能。

控制PPT翻页的手势可以定义为右手向右挥动，PPT播放下一页；左手向左挥动，PPT播放上一页。当然，在实现过程中还要考虑手势的歧义性，因为演讲者在演讲时常常会加入一些手势，要尽量避免设计的控制手势产生歧义，引发误操作。

该实现过程的前几步与前面章节的实例类似，这里仅作简单的介绍。

(1) 新建一个WPF工程，命名为KinectPowerPointControl。与前面的实例程序一样，添加Kinect程序集。

(2) 添加Window_Loaded()和Window_Closed()函数。在Window_Loaded()函数中声明kinectSensor对象并对其进行初始化设置，在Window_Closed函数中关闭Kinect设备，相关代

码如下：

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    kinectSensor = (from sensor in KinectSensor.KinectSensors
                    where sensor.Status == KinectStatus.Connected
                    select sensor).FirstOrDefault();
    kinectSensor.ColorStream.Enable(ColorImageFormat.RgbResolution640x480Fps30);
    kinectSensor.SkeletonStream.Enable();

    kinectSensor.Start();
    kinectSensor.ColorFrameReady += kinectSensor_ColorFrameReady;
    kinectSensor.SkeletonFrameReady += new EventHandler
        <SkeletonFrameReadyEventArgs>(kinectSensor_SkeletonFrameReady);
}
private void Window_Closed(object sender, EventArgs e)
{
    kinectSensor.Stop();
}
```



本实例用到了视频数据和骨骼追踪数据，其中骨骼追踪数据用来对演讲者进行姿势捕捉，而视频数据只是用来辅助测试姿势识别的结果。

(3) 界面准备。在MainWindow.xaml文件的设计器中添加一个Image控件来显示视频信息，再添加3个Ellipse控件来显示头部、左手和右手3个点的位置，具体代码如下：

```
<Grid>
    <Image Name="ColorImage"
           Width="640"
           Height="480"></Image>
    <Canvas Background="Transparent" Name="SkeletonCanvas">
        <Ellipse Fill="Red"
                 Height="20"
                 Width="20"
                 Name="ellipseLeftHand"
                 Stroke="White" />
        <Ellipse Fill="Red"
                 Height="20"
                 Width="20"
                 Name="ellipseRightHand"
                 Stroke="White" />
        <Ellipse Fill="Red"
                 Height="20"
                 Width="20"
                 Name="ellipseHead"
                 Stroke="White" />
     </Canvas>
</Grid>
```

(4) 添加`kinectSensor_ColorFrameReady()`事件处理函数，用以显示Kinect捕捉的视频数据，其内容与之前的实程序类似，如下所示：

```
private void kinectSensor_ColorFrameReady(object sender,
    ColorImageFrameReadyEventArgs e)
```

```
{
    using (ColorImageFrame imageFrame = e.OpenColorImageFrame())
    {
        if (imageFrame != null)
        {
            this.pixelData = new byte[imageFrame.PixelDataLength];
            imageFrame.CopyPixelDataTo(this.pixelData);
            this.ColorImage.Source =
                BitmapSource.Create(imageFrame.Width, imageFrame.Height, 96, 96,
                                    PixelFormats.Bgr32, null, pixelData,
                                    imageFrame.Width * imageFrame.BytesPerPixel);
        }
    }
}
```

(5) 添加`kinectSensor_SkeletonFrameReady()`事件处理函数，此函数获取骨骼数据的过程与实例3的相关函数类似，这里就不再赘述了。当正确捕捉到有效的骨骼数据后，调用`ProcessGesture()`函数进行手势的识别和判定，相关代码如下：

```
private void kinectSensor_SkeletonFrameReady(object sender,
    SkeletonFrameReadyEventArgs e)
{
    using (SkeletonFrame skeletonFrame = e.OpenSkeletonFrame())
    {
        if (skeletonFrame != null)
        {
            skeletonData =
                new Skeleton[kinectSensor.SkeletonStream.FrameSkeletonArrayLength];
            skeletonFrame.CopySkeletonDataTo(this.skeletonData);
            Skeleton skeleton = (from s in skeletonData
                where s.TrackingState == SkeletonTrackingState.Tracked
                select s).FirstOrDefault();
            if (skeleton!=null)
            {
                SkeletonCanvas.Visibility = Visibility.Visible;
                ProcessGesture(skeleton);
            }
        }
    }
}
```

(6) `ProcessGesture()`函数负责识别演讲者右手向右挥动和左手向左挥动的手势，并触发幻灯片的翻页操作。识别该手势用到了3个骨骼点，分别是头部节点、左手节点和右手节点。该函数的代码如下：

```
private void ProcessGesture(Skeleton s)
{
    Joint leftHand = (from j in s.Joints
                      where j.JointType == JointType.HandLeft
                      select j).FirstOrDefault();
    Joint rightHand = (from j in s.Joints
                       where j.JointType == JointType.HandRight
                       select j).FirstOrDefault();
```

```
    Joint head = (from j in s.Joints
                  where j.JointType == JointType.Head
                  select j).FirstOrDefault();

    if (rightHand.Position.X > head.Position.X + 0.45)
    {
        if (!isBackGestureActive && !isForwardGestureActive)
        {
            isForwardGestureActive = true;
            System.Windows.Forms.SendKeys.SendWait("{Right}");
        }
    }
    else
    {
        isForwardGestureActive = false;
    }
    if (leftHand.Position.X < head.Position.X - 0.45)
    {
        if (!isBackGestureActive && !isForwardGestureActive)
        {
            isBackGestureActive = true;
            System.Windows.Forms.SendKeys.SendWait("{Left}");
        }
    }
    else
    {
        isBackGestureActive = false;
    }
    //将头和左、右手的骨骼点与界面上对应的原点相关联
    SetEllipsePosition(ellipseHead, head, false);
    SetEllipsePosition(ellipseLeftHand, leftHand, isBackGestureActive);
    SetEllipsePosition(ellipseRightHand, rightHand, isForwardGestureActive);
}
```

先使用LINQ语句将这3个点从`Skeleton`集合中提取出来，然后利用这3个点的相对坐标信息判定演讲者此时的手势。

若右手节点的$x$坐标值比头部节点的$x$坐标值大0.45，则判定为右手向右挥动，此时`System.Windows.Forms.SendKeys.SendWait`函数发送单击右箭头的操作，从而实现幻灯片向下翻页；同理，若左手节点的$x$坐标值比头部节点的$x$坐标值小0.45，则判定为左手向左挥动，从而触发左箭头的按键操作，幻灯片就会向上翻页。

定义`isForwardGestureActive`和`isBackGestureActive`两个全局变量来辅助标定两个手势判定的开关。

(7) 上面用到的`SetEllipsePosition()`函数，主要负责将捕捉到的头和左、右手的点的位置赋值给3个Ellipse控件，其定义如下：

```
private void SetEllipsePosition(Ellipse ellipse, Joint joint, bool isHighlighted)
{
     ColorImagePoint colorImagePoint = kinectSensor.CoordinateMapper.
         MapSkeletonPointToColorPoint(joint.Position, ColorImageFormat.
         RgbResolution640x480Fps30);
```

5

```
        if (isHighlighted)
        {
            ellipse.Width = 60;
            ellipse.Height = 60;
            ellipse.Fill = Brushes.Red;
        }
        else
        {
            ellipse.Width = 20;
            ellipse.Height = 20;
            ellipse.Fill = Brushes.Blue;
        }
        Canvas.SetLeft(ellipse, colorImagePoint.X - ellipse.ActualWidth / 2);
        Canvas.SetTop(ellipse, colorImagePoint.Y - ellipse.ActualHeight / 2);
    }
```

该函数的第3个参数`isHighlighted`表示手势的位置，当右手向右挥动或左手向左挥动的时候，对应于右手或左手的圆点会变大，颜色由蓝色变为红色。

连接Kinect，运行实例程序。站在Kinect前方，可以在程序界面中看到标定头部、左手和右手的圆点。右手向右挥动，对应的圆点会变大，颜色由蓝色变为红色；左手向左挥动，对应的圆点也会变大，颜色由蓝色变为红色。打开PPT并放映，站在Kinect前方，右手向右挥动，幻灯片会播放下一页；左手向左挥动，幻灯片会播放上一页。这样，无需借助鼠标和键盘即可实现幻灯片的切换操作。

## 5.7　小结

本章主要介绍了骨骼追踪数据的结构，并结合实例3讲解了骨骼数据的获取和处理方式，实例4通过Kinect实现了对PPT播放的控制，这能够帮助读者更形象地理解如何利用骨骼追踪数据实现体感应用程序。另外，本章所介绍的半身模式和骨骼点旋转信息是Kinect最新版的SDK中新增的特性，半身模式支持只需上体姿势控制的应用；骨骼点旋转信息的加入有效地提高了特定姿势识别的准确度，这使得应用都更具真实感。

第6章

# 音频API的使用

语音识别的任务就是使用计算机程序将语音转换成一串词语。语言是人类最常用的沟通形式，而语音识别的终极目标就是能让人机之间更加自然、有效地进行交互。50年来，快速而准确的语音识别一直是计算机科学发展的目标之一。早期的技术试图对人类大脑理解声音和语言的方式进行建模，并尝试将这种模型植入语音识别器。但是这种方法以失败告终，这主要是因为人类大脑对语音的理解方式至今仍有大部分是未知的，并且计算机的处理能力十分局限。然而目前，随着计算机处理能力的巨大改进以及庞大的声音采样字典，语音识别技术已经可以在普通电脑上达到相当准确和稳定的效果。

本章主要介绍了Kinect语音识别的相关结构和功能，并结合两个实例程序讲解了如何通过调用Kinect for Windows SDK提供的语音API来实现相应的功能。



## 6.1 关于 Kinect 麦克风阵列

前面的章节在介绍Kinect的结构时，曾提到过Kinect传感器下方的四元麦克风阵列，它提供了语音识别功能的支持。与单独的麦克风相比，四元麦克风阵列在性能上具备以下显著优势。

- 提高音频质量。相对于单独的麦克风，麦克风阵列支持更加精细、高效的噪音抑制和自动回声消除算法。
- 波束成型和音频源定位。利用来自特定音频源的声音到达阵列中每个麦克风的微小时间差，结合波束成型技术，应用程序就可以确定音频源的方向，并且使用麦克风阵列作为可控的定向传声器。

凭借麦克风阵列优越的性能，再结合Kinect for Windows SDK，我们就可以开发出具有以下功能的应用程序：

- 高质量音频的捕获；
- 波束成型和音频源定位；
- 语音识别。

其中语音识别是NUI的一个关键因素，在Windows上由Microsoft Speech平台支持。Kinect for Windows SDK为托管的应用程序结合Microsoft Speech API使用Kinect麦克组提供了必需的基础架构，该架构支持最新的语音算法。该SDK还包括一个专门为Kinect传感器的麦克组定制的语音模型。

此外，Kinect for Windows SDK还提供了用于捕捉和控制音频的托管API。该API主要是在Kinect Audio DMO上的一个封装，与其支持同样的功能，但是用起来会更加简单。托管API允许应用程序配置DMO，并且执行启动、捕获和终止音频流的操作。托管API还包括向应用程序提供音频源和波束方向的事件函数。

## 6.2　实例 5——记录一段音频流，并监视音频源方向

本实例是一个C#控制台的应用程序，主要演示了如何从Kinect传感器的麦克风阵列捕获一段音频流，并监控音频源的方向。其基本流程如下：

(1) 创建一个对象，用以表示Kinect传感器的麦克风阵列；

(2) 捕获音频流，并将其写入一个文件；

(3) 监控音频源的方向。

下面详细介绍该应用程序的实现过程，具体步骤如下。

(1) 新建一个C#控制台应用项目，命名为“KinectRecordAudio”，如图6-1所示。

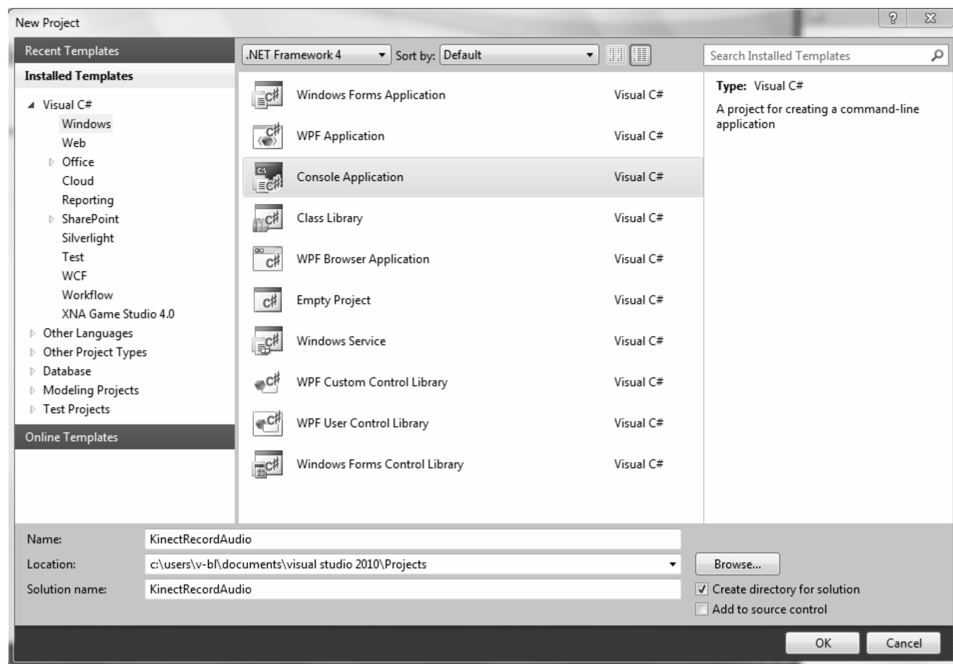



图6-1　新建控制台应用程序

(2) 与前面的实例一样，首先添加Kinect库，并在应用程序中引入Microsoft.Kinect命名空间；然后创建KinectSensor对象，将连接到电脑的Kinect设备赋值给该对象；接着调用Start函数启动Kinect设备。相关代码如下：

```
static void Main(string[] args)
{
    KinectSensor kinectSensor = (from k in KinectSensor.KinectSensors
                                 where k.Status == KinectStatus.Connected
                                 select k).FirstOrDefault();
    if (kinectSensor == null)
    {
        Console.WriteLine("No Kinect Connected!\n"+
            "Press any Key to continue.\n");
        Console.ReadKey(true);
        return;
    }

    try
    {
        kinectSensor.Start();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.ToString() + "\nPress any key to continue.\n");
        Console.ReadKey(true);
        return;
    }
    const int voiceRecordingTime = 20;
    const int voiceRecordingLength = voiceRecordingTime * 2 * 16000;
    const string outputFileName = "voiceRecord.wav";
    var voiceBuffer = new byte[4096];
    ……
}
```

6

上述代码中，4个控制音频处理的常量含义如下所示。

- voiceRecordTime，记录时间设置为20秒；
- voiceRecordingLength，表示记录长度，以字节为单位，其值为记录时间×采样大小（2字节）×每个采样的比特数（16000）。
- outputFileName，写入音频数据的文件名，程序执行完毕后，该音频文件保存在工程目录的Debug文件夹下。
- voiceBuffer，读写音频流的缓冲数组，大小为4096B；

(3) 接下来，在Main函数里创建并配置一个Kinect音频源对象，以获取Kinect设备捕获到的音频源，相关代码如下：

```
KinectAudioSource audioSource = kinectSensor.AudioSource;
audioSource.AutomaticGainControlEnabled = false;
Thread.CurrentThread.Priority = ThreadPriority.Highest;

audioSource.BeamAngleChanged +=new
    EventHandler<BeamAngleChangedEventArgs>(audioSource_BeamAngleChanged);
```

创建一个KinectAudioSource类型的对象audioSource，用来表示Kinect传感器的声音信息。KinectSensor的KinectAudioSource类型的AudioSource成员保存了获取到的音频流，直接将它赋值给audioSource。

audioSource的AutomaticGainControlEnabled属性决定了Kinect Audio DMO是否执行自动增益控制，本实例设置为false，其他属性值均使用默认值。

实例程序还通过设置Thread.CurrentThread.Priority为ThreadPriority.Highest，将本程序的优先级设置为最高，这样能有效地避免丢失采样。

最后，订阅KinectAudioSource的BeamAngleChanged事件处理函数。当波束角方向改变时，应用程序便会触发相应的事件处理函数。

(4) 记录音频流。RecordAudio开启音频流，记录20秒，并将记录的音频流写入一个.wav文件，相关代码如下：

```
static void Main(string[] args)
{
    ……
    using (var fileStream = new FileStream(outputFileName, FileMode.Create))
    {
        WriteWavHeader(fileStream, voiceRecordingLength);
        Console.WriteLine("Recording for {0} seconds...", voiceRecordingTime);

        using (var audioStream = audioSource.Start())
        {
            int count = 0;
            int totalCount = 0;

            while (totalCount < voiceRecordingLength && (count =
                 audioStream.Read(voiceBuffer , 0, voiceBuffer .Length)) > 0)
            {
                fileStream.Write(voiceBuffer , 0, count);
                totalCount += count;

                if (audioSource.SoundSourceAngleConfidence > 0.85)
                    Console.Write("Sound source position (radians):{0}\t\tBeam:{1}\r",
                        audioSource.SoundSourceAngle, audioSource.BeamAngle);
            }
        }
    }
}
```

在开始记录之前，先创建一个FileStream对象来表示输出文件，并且调用私有方法WriteWavHeader写入文件的.wav头。WriteWavHeader方法的具体内容请参阅本书附带的源代码。

接着调用KinectAudioSource.Start函数开启音频流，并返回关联的Stream对象。音频流的记录操作在while循环中进行，先调用Stream.Read函数将音频流读取到缓存中，然后调用FileStream.Write函数将缓存写入到输出文件中。KinectAudioSource类的SoundSource-AngleConfidence属性表示音频源位置估计的置信度，如果置信度的值大于0.85，循环中就会打印出音频源和波束的方向。当记录的缓存数量到达指定的记录长度，循环则会终止。

(5) 监视波束方向的变化。当KinectAudioSource的BeamAngle属性发生变化时会触发BeamChanged事件，在它的事件处理函数audioSource_BeamAngleChanged中打印出新的波束方向。相关程序代码如下：

```
private static void audioSource_BeamAngleChanged(object sender,
```

```
        BeamAngleChangedEventArgs e)
    {
        Console.WriteLine("Beam Angle Changed:{0}", e.Angle);
    }
```

BeamAngleChangedEventArgs对象e的Angle属性表示当前的波束角度，单位为弧度。从用户面对Kinect传感器的透视方向来看，该角度的说明如下：

- 0表示波束径直在传感器前；
- 正角度表示波束在传感器的右边；
- 负角度表示波束在传感器的左边。

(6) 运行示例程序。按下Ctrl+F5组合键运行程序，来回走动着说话，测试输出结果如图6-2所示。

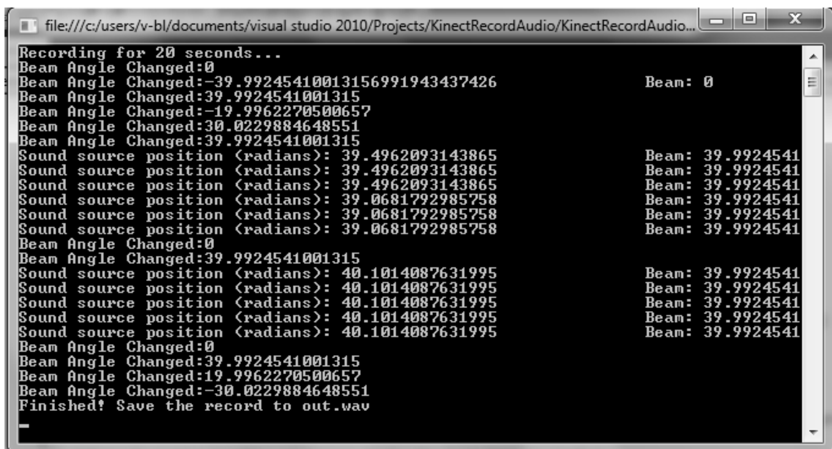



图6-2　运行结果

## 6.3　实例 6——调用语音 API，实现语音识别小程序

本实例是一个C#控制台应用程序，主要演示了如何使用Kinect for Windows SDK的语音识别功能。编写语音识别程序还需下载安装微软语言识别平台开发工具包（Microsoft Speech Platform SDK），下载地址为http://www.microsoft.com/en-us/download/details.aspx?id=27226。

本实例程序的基本流程如下所示：

(1) 创建一个对象，表示Kinect传感器的麦克风阵列；

(2) 创建一个语音识别的对象，并设定语法；

(3) 对命令进行响应。

下面详细说明该应用程序的实现步骤。

(1) 新建一个C#控制台应用的项目，命名为“KinectSpeech”，如图6-3所示。

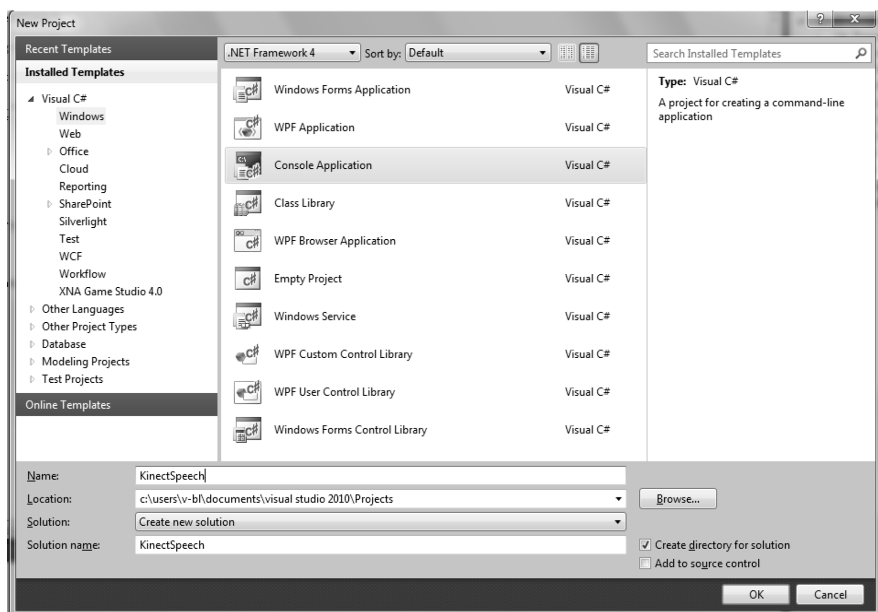



图6-3　新建控制台应用程序

(2) 与前面的实例一样，首先添加`Kinect`库，并在应用程序中引入`Microsoft.Kinect`命名空间；然后创建`KinectSensor`对象，获取连接到电脑的Kinect设备并将其赋值给该对象；接着调用`Start`函数启动Kinect设备。相关代码如下：

```
public static void Main(string[] args)
{
    KinectSensor KinectSensor = (from k in KinectSensor.KinectSensors
                                 where k.Status == KinectStatus.Connected
                                 select k).FirstOrDefault();
    if (KinectSensor == null)
    {
        Console.WriteLine("No Kinect Connected\n" + "Press any key to continue.\n");
        Console.ReadKey(true);
        return;
    }
    KinectSensor.Start();
    ……
}
```

(3) 添加语音识别库。右击“References”，在弹出的快捷菜单中选择“Add reference…”菜单项，找到Microsoft Speech Platform SDK安装的目录，默认目录为“C:\Program Files\Microsoft SDKs\Speech\v11.0\Assembly”，将目录下的“Microsoft.Speech.dll”添加进来，并在代码头添加该命名空间的引用，如下所示：

```
using Microsoft.Speech.AudioFormat;
using Microsoft.Speech.Recognition;
```

(4) 接着，在Main函数里创建和配置一个Kinect音频源对象，以获取Kinect设备捕获到的音频源，相关代码如下：

```
KinectAudioSource audioSource = KinectSensor.AudioSource;
audioSource.EchoCancellationMode = EchoCancellationMode.None;
audioSource.AutomaticGainControlEnabled = false;
```

创建一个KinectAudioSource类型的对象audioSource，用来表示Kinect传感器的声音信息，KinectSensor的KinectAudioSource类型的AudioSource成员保存了获取到的音频流，直接将它赋值给audioSource。

将audioSource的EchoCancellationMode设置为EchoCancellationMode.None，即不在本实例程序中采用回波消除和抑制。audioSource的AutomaticGainControlEnabled属性决定了Kinect Audio DMO是否执行自动增益控制，本实例设置为false，即不采用自动增益控制。其他属性值均使用默认值。

(5) 在Speech中创建一个语音识别引擎，相关代码如下：

```
RecognizerInfo recognizerInfo = GetKinectRecognizer();

private static RecognizerInfo GetKinectRecognizer()
{
    Func< RecognizerInfo, bool > matchingFunc = r =>
    {
        string value;
        r.AdditionalInfo.TryGetValue("Kinect", out value);
        return "True".Equals(value, StringComparison.InvariantCultureIgnoreCase)
            && "en-US".Equals(r.Culture.Name,
            StringComparison.InvariantCultureIgnoreCase);
    };
    return SpeechRecognitionEngine.InstalledRecognizers().Where(matchingFunc).
        FirstOrDefault();
}
```

SpeechRecognitionEngine类提供了一系列获取和管理语音识别引擎的方法，比如接下来要用到的加载语法器、开始执行语音识别、结束语音识别等。其中的InstalledRecognizers是静态方法，它会返回一个语音识别器的列表，该列表包括了当前系统上安装的所有语音识别器。Speech使用LINQ语句获取列表中第一个识别器的ID，并将结果作为RecognizerInfo的对象返回，RecognizerInfo类用以表示语音识别引擎对象的相关信息，如Name、Id、Culture等。接下来，Speech将使用RecognizerInfo.Id创建一个SpeechRecognitionEngine对象：

```
var speechRecognitionEngine = new SpeechRecognitionEngine(recognizerInfo.Id)
```

(6) 指定语音命令。本实例设计了3个语音命令：red、green和blue。要指定这些命令，需要创建并加载一个包含要识别的词语的语法器，具体代码如下：

```
using (var speechRecognitionEngine = new SpeechRecognitionEngine(recognizerInfo.Id))
{
    var colors = new Choices();
    colors.Add("red");
    colors.Add("green");
```

```
    colors.Add("blue");

    var grammatBuilder = new GrammarBuilder { Culture = recognizerInfo.Culture };
    grammatBuilder.Append(colors);
    var g = new Grammar(grammatBuilder);

    speechRecognitionEngine.LoadGrammar(g);
    speechRecognitionEngine.SpeechRecognized += SreSpeechRecognized;
    speechRecognitionEngine.SpeechHypothesized += SreSpeechHypothesized;
    speechRecognitionEngine.SpeechRecognitionRejected +=
        SreSpeechRecognitionRejected;
    ……
}
```

Choices对象表示要识别的语句列表。通过调用Choices.Add方法可以将要识别的语句添加到列表中。完成该列表后，接着创建一个GrammarBuilder对象，它能够以简单的方式构造一种语法，并且保证其语义与识别器的语义匹配。然后将Choices对象传递给GrammarBuilder.Append方法，从而定义语法元素。最后，通过调用SpeechRecognitionEngine.LoadGrammar方法，将定义好的语法元素加载到语音识别引擎中。

每当用户说出一个单词，语音识别便会在语法器的单词模板中对用户语音进行比对，从而确定该语音是否为需要被识别的命令。然而，语音识别是一种固有的模糊处理方式，因此每一次识别都伴随一个估计的置信度值。

Speech的引擎会触发以下3个事件。

- SpeechRecognitionEngine.LoadGrammar事件会在每次尝试命令时发生。它会传递给事件处理函数一个SpeechRecognizedEventArgs对象，该对象包含一个从命令集合中选出的最佳匹配单词和一个估计的置信值。
- SpeechRecognitionEngine.SpeechRecognized事件会在尝试的命令被识别为命令集合中的成员时发生。该事件会传递给事件处理函数一个包含识别出的命令的SpeechRecognizedEventArgs对象。
- SpeechRecognitionEngine.SpeechRejected事件会在尝试的命令未被识别为命令集成员时发生。它会传递给事件处理函数一个SpeechRecognitionRejectedEventArgs对象。

在实例程序中将这3个事件全部注册，并且实现相应的操作，具体代码如下：

```
private static void SreSpeechRecognitionRejected(object sender,
    SpeechRecognitionRejectedEventArgs e)
{
    Console.WriteLine("\nSpeech Rejected");
    if (e.Result != null)
    {
        DumpRecordedAudio(e.Result.Audio);
    }
}
```

```
private static void SreSpeechHypothesized(object sender, SpeechHypothesizedEventArgs e)
{
    Console.Write("\rSpeech Hypothesized: \t{0}", e.Result.Text);
}

private static void SreSpeechRecognized(object sender, SpeechRecognizedEventArgs e)
{
    if (e.Result.Confidence >= 0.7)
    {
        Console.WriteLine("\nSpeech Recognized: \t{0}\tConfidence:\t{1}",
            e.Result.Text, e.Result.Confidence);
    }
    else
    {
        Console.WriteLine("\nSpeech Recognized but confidence was too low: \t{0}",
            e.Result.Confidence);
        DumpRecordedAudio(e.Result.Audio);
    }
}
```

前两个操作只是简单地打印出来自事件对象的关键数据。SreSpeechRecognitionRejected处理程序调用私有的DumpRecordedAudio方法，将记录的单词写入WAV文件。

(7) 识别命令。在语音识别配置完成后，Speech将启动处理流程。语音识别引擎会自动识别出语法中的单词，并且根据识别的情况触发相应的事件，相关代码如下：

```
using (Stream s = audioSource.Start())
{
    speechRecognitionEngine.SetInputToAudioStream(
        s, new SpeechAudioFormatInfo(EncodingFormat.Pcm,
        16000, 16, 1, 32000, 2, null));

    Console.WriteLine(
        "Recognizing speech. Say: 'red', 'green' or 'blue'. Press ENTER to stop");

   speechRecognitionEngine.RecognizeAsync(RecognizeMode.Multiple);
   Console.ReadLine();
   Console.WriteLine("Stopping recognizer ...");
   speechRecognitionEngine.RecognizeAsyncStop();
}
```

Speech通过调用KinectAudioSource.StartCapture，从Kinect传感器的麦克风阵列捕获音频。然后进行以下操作。

① 调用SpeechRecognitionEngine.SetInputToAudioStream指定音频源及其特征。

② 调用SpeechRecognitionEngine.RecognizeAsync指定异步的识别。语音识别引擎会一直在后台运行，直至用户通过按键终止该进程。

③ 调用SpeechRecognitionEngine.RecognizeAsyncStop停止识别处理并关闭引擎。

(8) 运行示例程序

按下Ctrl+F5组合键运行程序，面向Kinect说出red、green或blue，程序会对应每个命令打印出相应的通知，主要包含以下信息：

- ❑ 命令集合中的哪一个成员最接近说出的命令；
- ❑ 评估的置信度值；
- ❑ 该命令是否被识别为命令集中的成员。

测试输出结果如图6-4所示。

```
file:///C:/Users/v-bl/Documents/C#/Speech/Bin/Debug/Speech.EXE
Using: Microsoft Server Speech Recognition Language - Kinect (en-US)
Recognizing speech. Say: 'red', 'green' or 'blue'. Press ENTER to stop
Speech Hypothesized:    red
Speech Recognized but confidence was too low:   0.2764376

Writing file: RetainedAudio_6.wav
Speech Hypothesized:    blue
Speech Recognized but confidence was too low:   0.2628398

Writing file: RetainedAudio_7.wav
Speech Hypothesized:    blue
Speech Recognized but confidence was too low:   0.4570868

Writing file: RetainedAudio_8.wav
Speech Hypothesized:    green
Speech Recognized but confidence was too low:   0.4486768

Writing file: RetainedAudio_9.wav
Speech Hypothesized:    red
Speech Recognized but confidence was too low:   0.5912446

Writing file: RetainedAudio_10.wav
_
```

图6-4　运行结果

## 6.4　小结

尽管Kinect在语音识别的性能上有所提升，但是单纯依赖Kinect for Windows SDK提供的Speech API很难实现有效的语音识别。由于大多程序的应用场景都比较嘈杂，噪音较多，再加上发音的差异，因此应用程序很难准确地定位到玩家想要通过语音触发的功能上。一般而言，增加识别字符串的单词数量，可以提高语音识别的准确度，比如，使用“Kinect Play”就远比只用“Play”作为语音命令有效得多。还可以结合手势识别提高语音识别的精确度，因为Kinect并不知道什么时候才要进行语音识别，如果一直使语音识别程序处于运行状态，就有可能匹配到一段噪音，跳转到未知的处理流程上。这时就可以设计一个手势，只有当Kinect捕捉到这一特定的手势时才开启语音识别，否则保持关闭，这样可有效减少干扰。此外，目前语音识别对中文的支持有限，最好使用简单的英文作为语音命令。

第 7 章

# Kinect for Windows Developer Toolkit介绍

2012年5月21日，微软官方发布了Kinect SDK 1.5版本。相对于之前的1.0版本，它最大的改变在于增加了一个工具包——Kinect for Windows Developer Toolkit，其中主要包含Kinect Studio和Face Tracking SDK两个辅助工具，并附带了一些新的示例代码。另外，SDK中的骨骼跟踪功能也有了质的飞跃，可以实时获取骨骼点的旋转信息。在本章中，我们将介绍这些新的功能及其使用方法。

## 7.1 安装 Kinect for Windows Developer Toolkit



因为Kinect for Windows Developer Toolkit的安装包是和SDK分离的，所以在使用前，需要先对其进行安装。安装开发者工具包的前提是已经安装过1.5或以上版本的Kinect for Windows SDK，这一部分内容请参阅第4章。下面仅介绍开发者工具包的安装方法。

Kinect for Windows官方网站提供了开发者工具包安装包的下载（http://www.microsoft.com/en-us/kinectforwindows/develop/developer-downloads.aspx），如图7-1所示。

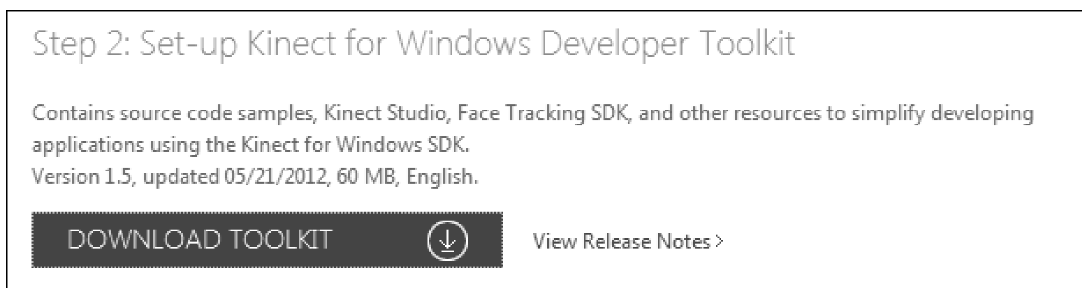



图7-1　Kinect官网上的下载页面

单击“DOWNLOAD TOOLKIT”下载安装包，并运行。首先依旧是最终用户许可协议，仔细阅读后勾选“我同意许可条款和条件”并单击“安装”按钮，如图7-2所示。

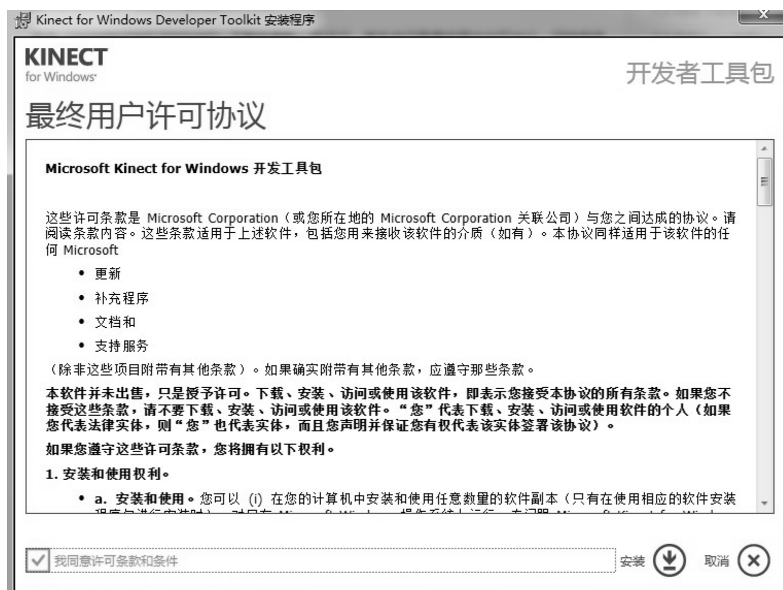



图7-2　开发者工具包用户协议

然后等待安装结束即可。当看到“安装已完成”时，如图7-3所示，说明开发者工具包已经安装成功了，可以开始下一步的功能体验了。




图7-3　开发者工具包安装完成

## 7.2 Kinect Studio 简介

在开发Kinect应用时，我们经常会遇到这样一种场景：写完一段功能代码，对其进行测试时，需要开发者运行程序并且从座位上站起来，走到Kinect前，通过相应的动作和手势才能完成测试，最后再回到电脑前继续写代码。这个过程如果重复多次，就会让人感觉极其麻烦，而如果在测试的同时需要查看相应的参数，甚至还需要其他人的帮忙。这对开发者而言简直是一场噩梦，而Kinect Studio就是用来解决此类问题的。

Kinect Studio可以像摄像机一样，记录原始的深度和彩色数据流，并在Kinect应用中重放。如此一来开发者就可以使用提前记录的一段数据流，坐在电脑前调试代码了，这将极大地提高Kinect程序员的开发效率。下面我们就来介绍如何使用Kinect Studio的强大功能。

### 7.2.1 打开 Kinect Studio 并链接应用

在“开始”菜单中找到Kinect Studio并打开，如图7-4所示。

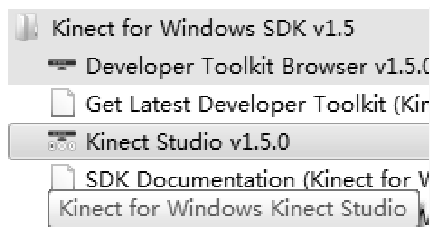



图7-4 打开Kinect Studio

单击运行后，我们就可以看到Kinect Studio的主界面，如图7-5所示。

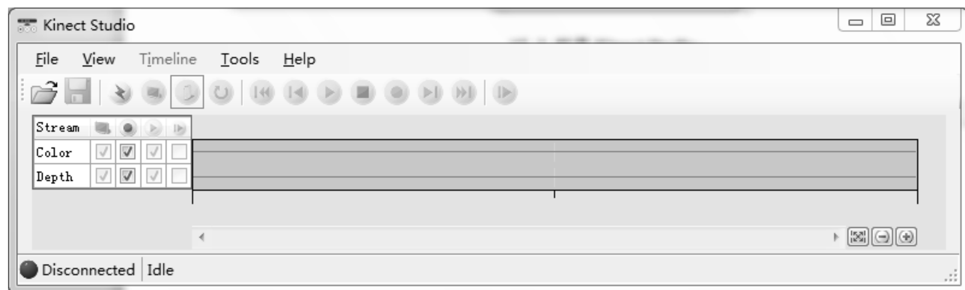



图7-5 Kinect Studio主界面

同时还会弹出一个连接应用的对话框，如图7-6所示。

这是因为Kinect Studio是Kinect应用的辅助工具，记录数据之前需要先将其连接到一个已经打开的Kinect应用上。我们打开最基础的“Skeletal Viewer”实例，并单击连接对话框上的“Refresh”按钮，就可以看到这个应用了，如图7-7所示。

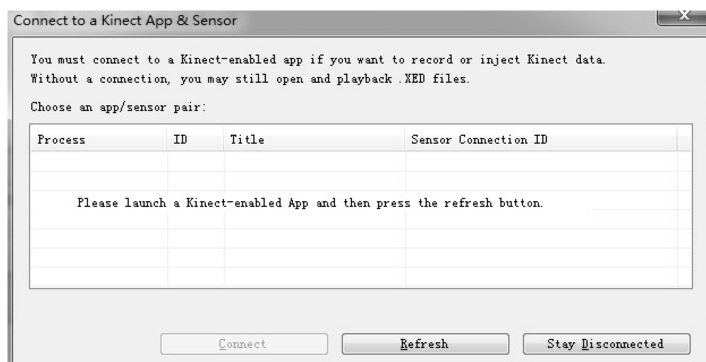



图7-6　Kinect Studio连接应用

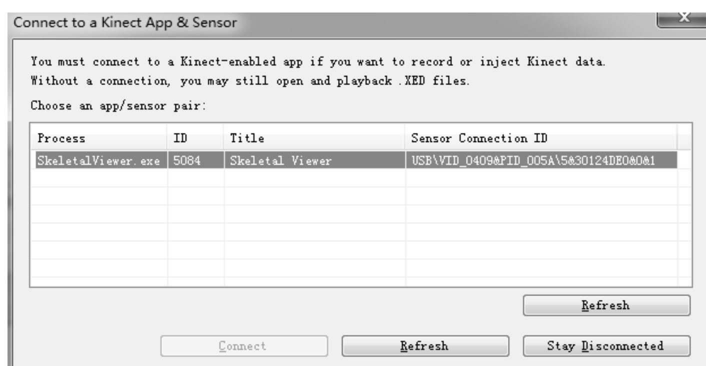



图7-7　刷新后的连接应用对话框

最后，单击“Connect”按钮就可以连接到Skeletal Viewer应用上了。

## 7.2.2　记录并回放 Kinect 数据流

本节将介绍如何使用Kinect Studio完成最基本的调试辅助功能——记录和回放。

### 1. 记录调试动作

首先应在左侧的Stream面板中，选中Color和Depth的记录属性，如图7-8所示。




图7-8　Stream面板

然后单击工具栏中的记录按钮（ ），这时记录已经开始，开发者可以在Kinect前做相应的调试动作。动作完成后，单击工作栏中的停止按钮（ ）就可以完成记录工作了，如图7-9所示。

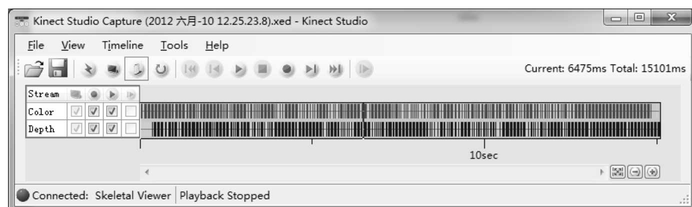



图7-9　记录数据后的Kinect Studio面板

#### 2. 在应用中回放记录的动作

通过上述操作，我们已经完成了Kinect数据流的记录工作，接下来就可以使用回放功能了。这里需要注意的是，工具栏上的应用调试按钮（ ）已经处于按下的状态，此时进行回放会将刚才记录的数据直接输入到Skeletal Viewer应用中。单击回放按钮（ ）开始回放，Skeletal Viewer应用将重复播放刚才记录的那段动作，如图7-10所示。

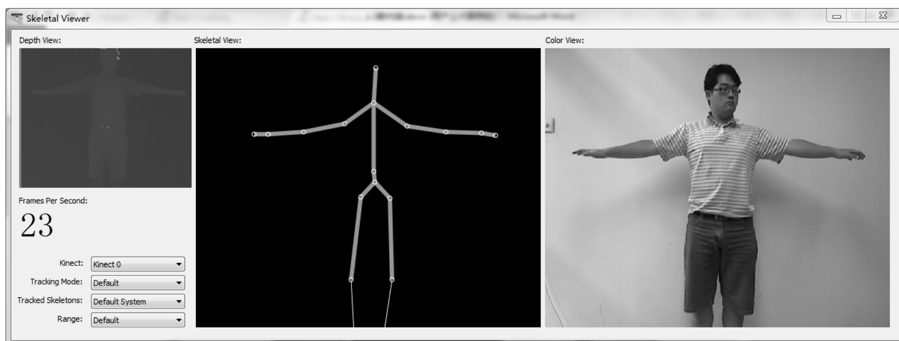



图7-10　在Skeletal Viewer中回放动作

不仅如此，Kinect Studio还支持单帧回放的功能，其中各个按钮的作用如下所示。

- 初始帧（ ）：跳转到记录的第一帧；
- 上一帧（ ）：返回到上一帧的图像；
- 下一帧（ ）：前进到下一帧的图像；
- 结束帧（ ）：跳转到记录的最后一帧。

#### 3. 在数据查看窗口中查看回放动作

虽然Skeletal Viewer应用可以直观地显示Kinect的彩色、深度数据流，但并不是所有的应用都有这样的窗口。为此，Kinect Studio还提供了专门的窗口，可以让开发者直观地查看原始数据流。

首先在“View”菜单选中“Color”、“Depth”和“3D View”选项，如图7-11所示。

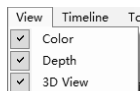



图7-11　在View菜单中选中相应的选项

这时就会出现对应的3个窗口，和上面提到的回放方法一样，单击回放和单步回放按钮，即可查看记录的数据，如图7-12所示。其中，第一个窗口显示的是深度图和彩色图的组合视图，第二个窗口显示的是彩色图，第三个窗口显示的是深度图。




图7-12　3个对应的数据查看窗口

### 7.2.3　保存和载入 Kinect 数据流

由于之前记录和回放操作的数据都保存在内存中，因此我们只能在当前的电脑中进行调试，无法将其复制到其他地方。为此，Kinect Studio提供了保存和载入的功能，可以将这些数据写到硬盘文件中，这样我们就可以在多台电脑上调试同一个动作了。

主面板工具栏最左边的两个按钮就是载入和保存按钮。单击保存按钮，将会弹出保存对话框，选择保存路径和文件名进行保存，如图7-13所示。




图7-13　Kinect Studio保存对话框

值得注意的是，Kinect原始数据流的数据量很大，一段时长15秒的数据保存成文件就能达到300MB。因此在保存时，应尽量选取其中的主要调试动作，这样可以减轻硬盘的负担。

在时间轴上选取需要保存的数据段，这些段会以黄色显示。然后在“Timeline”菜单中选择“Save Range As”菜单项，即可将选取的数据段保存成文件，如图7-14所示。

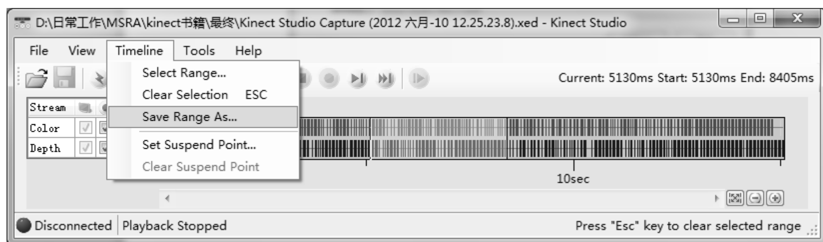



图7-14　保存选定的数据段

## 7.3　Face Tracking SDK 简介

Toolkit中另一个引人注目的功能是Face Tracking SDK。顾名思义，它是用来追踪并提供详细面部信息的。前面已经提到过，Kinect SDK提供的骨骼跟踪较为粗糙，只是将头部识别成单一的骨骼点，这样仅能得到头的位置，而无法获取面部朝向和表情等特征，Face Tracking则解决了这些问题。本节将详细介绍Face Tracking SDK的功能和用法，下一节我们通过一个实例进行简单演示。



### 7.3.1　Face Tracking SDK 主要功能

我们先来看一个使用Face Tracking SDK开发的趣味应用，如图7-15所示。

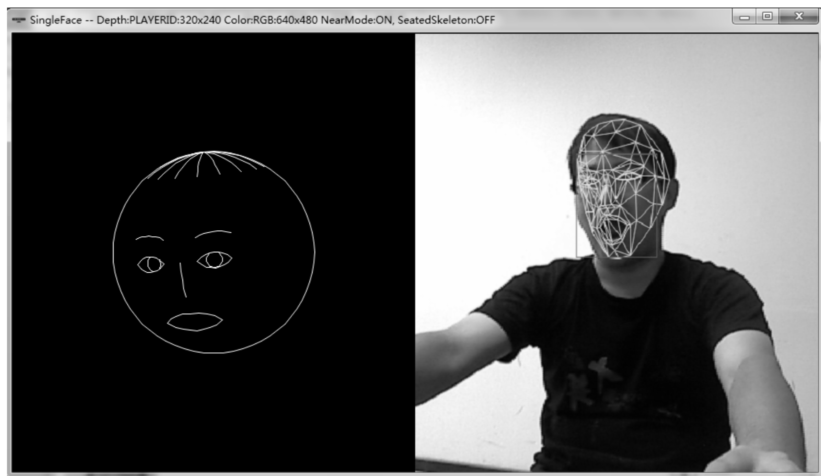



图7-15　Face Tracking趣味应用

这个应用可以将右侧真实的人脸图像转变为一副抽象的人脸图像，其中脸的朝向、嘴型和眉型都能实时地和真实人脸对应起来。它就是根据Face Tracking识别的人脸模型构造出来的。因此在使用Face Tracking之前，我们首先要了解这个库究竟能做什么，即能识别出人脸的哪些数据。Face Tracking可以识别出的人脸数据主要包括以下几项：

- 特征点坐标；
- 面部朝向；
- 包围盒；
- 基于Candide3人脸模型的参数。

下面分别对这几项内容进行介绍。

**1. 特征点坐标**

Face Tracking可以根据Kinect提供的深度图和彩色图，对人脸的100个特征点进行识别和追踪，如图7-16所示。

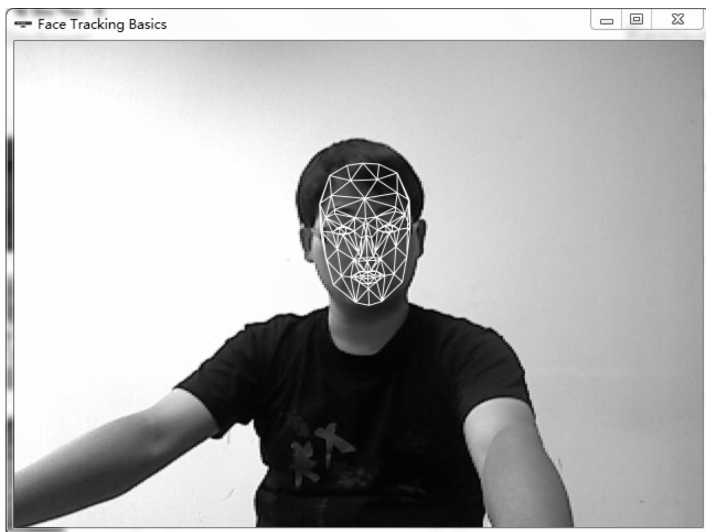



图7-16　Face Tracking识别的人脸特征点

图中的白色“面具”就是由Face Tracking识别出的一部分特征点组成的。通过调用`FaceTrackFrame.GetProjected3DShape()`和`FaceTrackFrame.Get3DShape()`两个函数，可以获得这些特征点的平面坐标和立体坐标。

**2. 面部朝向**

在Kinect SDK获取的头部位置的基础上，Face Tracking还可以获得头部的转动数据，即面部朝向，可以通过`FaceTrackFrame.Rotation()`函数实现。

**3. 包围盒**

为了便于开发者处理自定义的面部信息，Face Tracking还提供了人脸的矩形包围盒数据，通

过`FaceTrackFrame.FaceRect`返回结果。

4. 基于Candide3人脸模型的参数

为了更加简便地将人脸识别的结果应用到动画制作以及模型控制上，Face Tracking还返回了人脸基于Candide3模型的参数。Candide3模型是一种通用的人脸模型，通过少量的参数就可以表现出人脸的不同形态以及表情，很多3D模型都是用它来模拟人脸的。调用Face Tracking中的`FaceTrackFrame.GetAnimationUnitCoefficients()`函数可以直接获得这些参数。

### 7.3.2 Face Tracking SDK 使用方法

在了解了Face Tracking的功能后，剩下的问题就是如何在现有的Kinect SDK上使用它的功能。

我们首先应该明确一点，Face Tracking是一个基于Kinect SDK的扩展类库，其使用方法类似于Kinect Toolbox和Coding4Fun等第三方类库。也就是说，需要在初始化Kinect数据的基础上，将每一帧的彩色图、深度图和骨骼数据传递给Face Tracking，之后它才会返回这些帧的识别结果，包含上面提到的各种数据，如图7-17所示。

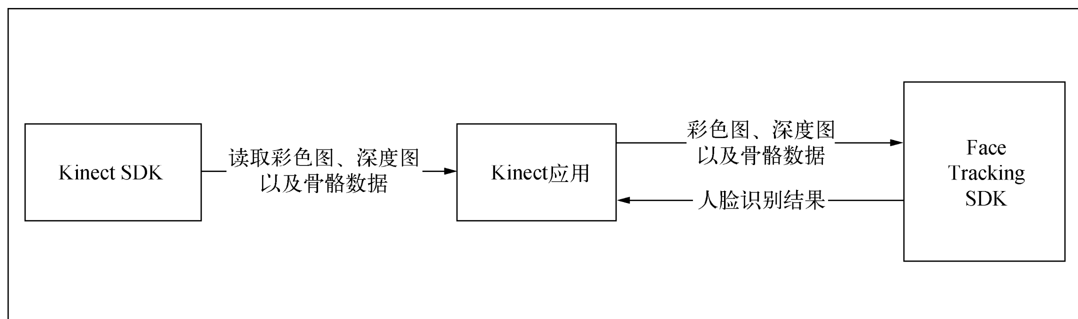



图7-17 Face Tracking的调用方法

## 7.4 实例 7——使用 Face Tracking SDK 识别人脸

前面介绍了Face Tracking的主要功能及其调用方法，本节将通过编写一个简单的Face Tracking应用来演示它的具体使用方法。

### 7.4.1 新建项目并添加引用

在编写代码之前，我们要先找到Face Tracking的链接库。打开Developer Toolkit Browser，在“Components”标签中下载并安装“Microsoft.Kinect.Toolkit”和“Microsoft.Kinect.Toolkit.FaceTracking”，如图7-18所示。

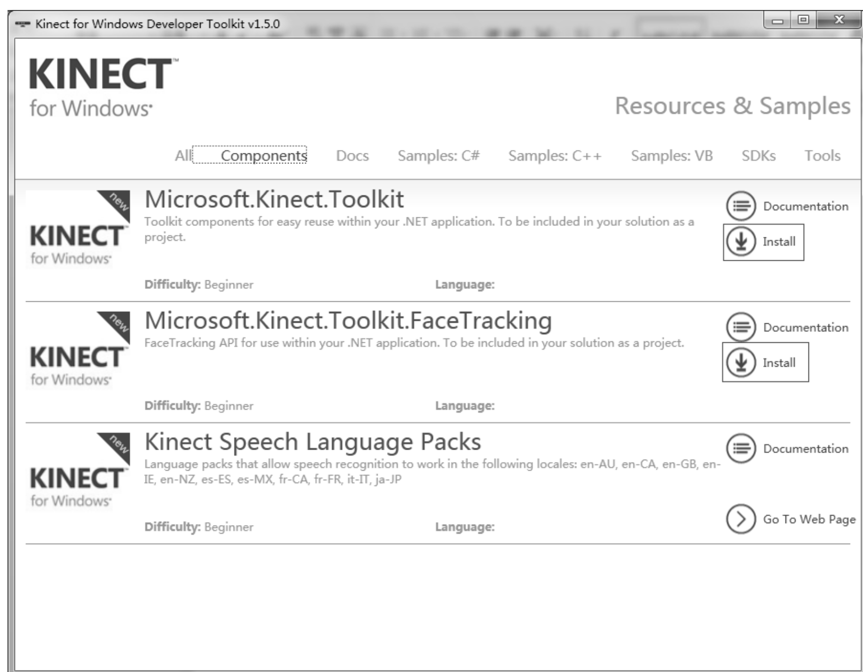



图7-18　安装FaceTracking

然后打开Visual Studio 2010，新建WPF项目，并命名为“FaceTest”，如图7-19所示。

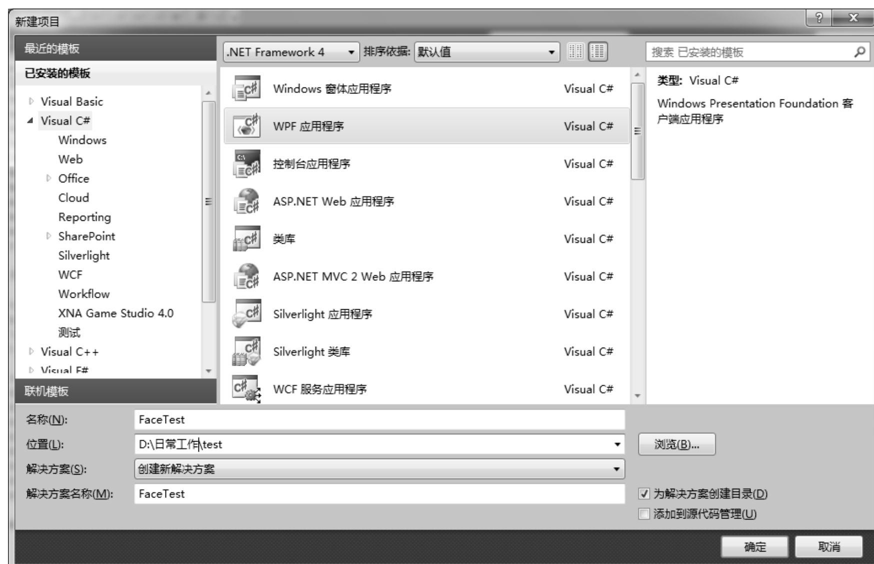



图7-19　新建FaceTest项目

接着在“解决方案资源管理器”中加入刚才安装的两个项目代码（单击鼠标右键，然后选择“添加”→“现有项目”选项，选择相应的.csproj文件），如图7-20所示。

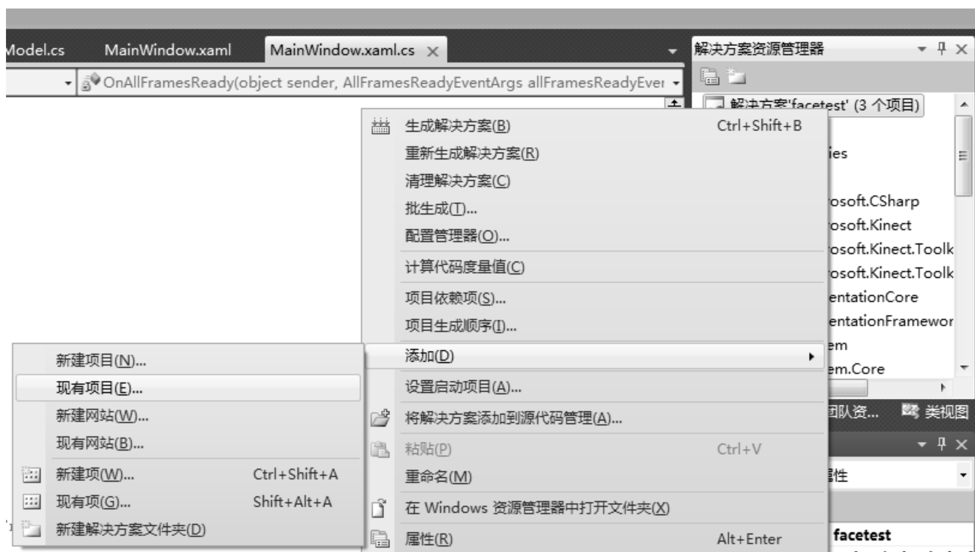


图7-20　在解决方案中添加已有的两个项目

最后，在主项目FaceTest中添加刚才加入的两个项目的引用，如图7-21所示。



图7-21　在项目中添加对Face Tracking SDK的引用

### 7.4.2　初始化 Kinect 数据流

前面已经提到过，Face Tracking的数据源是Kinect获取到的每一帧的彩色、深度以及骨骼数据，因此首先要用前面章节提到的方法初始化Kinect。相关代码如下：

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    kinectSensor = (from sensor in KinectSensor.KinectSensors
                    where sensor.Status == KinectStatus.Connected
                    select sensor).FirstOrDefault();
    kinectSensor.Start();
    kinectSensor.ColorStream.Enable(ColorImageFormat.
                                    RgbResolution640x480Fps30);
    kinectSensor.DepthStream.Enable(DepthImageFormat.
                                    Resolution320x240Fps30);
    try
    {
        kinectSensor.DepthStream.Range = DepthRange.Near;
        kinectSensor.SkeletonStream.EnableTrackingInNearRange = true;
    }
    catch (InvalidOperationException)
    {
        kinectSensor.DepthStream.Range = DepthRange.Default;
        kinectSensor.SkeletonStream.EnableTrackingInNearRange = false;
    }

    kinectSensor.SkeletonStream.TrackingMode =
        SkeletonTrackingMode.Seated;
    kinectSensor.SkeletonStream.Enable();
    kinectSensor.AllFramesReady += OnAllFramesReady;
}

private void OnAllFramesReady(object sender,
                              AllFramesReadyEventArgs
                              allFramesReadyEventArgs)
{
}
```

这里我们使用了近景以及坐姿模式，因此只需考虑面部的识别效果，可以忽略下半身的骨骼数据。接下来，我们就可以在`OnAllFramesReady()`函数中处理数据了。

### 7.4.3　获取数据并传入 Face Tracking

在调用Face Tracking之前，首先要定义一些数据成员用来存放Kinect获取的原始数据，相关代码如下：

```
private int trackedID;
private FaceTracker faceTracker;
private byte[] colorImage;
private ColorImageFormat colorImageFormat = ColorImageFormat.Undefined;
private short[] depthImage;
```

```
private DepthImageFormat depthImageFormat = DepthImageFormat.Undefined;
private bool disposed;
private Skeleton[] skeletonData;
```

在上面的代码中，colorImage、depthImage和skeletonData用来记录Kinect原始数据流；colorImageFormat和depthImageFormat用来防止出现格式问题；trackedSkeletons用来对每一个追踪到的骨骼进行人脸识别。FaceTracker是Face Tracking的核心类，类似于Kinect SDK中的KinectSensor，提供传入、传出数据的函数接口。

接下来，在OnAllFramesReady()函数中调用FaceTracker传入数据并获得结果。相关代码如下：

```
private void OnAllFramesReady(object sender,
    AllFramesReadyEventArgs allFramesReadyEventArgs)
{
    ColorImageFrame colorImageFrame = null;
    DepthImageFrame depthImageFrame = null;
    SkeletonFrame skeletonFrame = null;
    colorImageFrame = allFramesReadyEventArgs.OpenColorImageFrame();
    depthImageFrame = allFramesReadyEventArgs.OpenDepthImageFrame();
    skeletonFrame = allFramesReadyEventArgs.OpenSkeletonFrame();

    if (colorImageFrame == null || depthImageFrame == null || skeletonFrame == null)
    {
        return;
    }

    if (this.depthImageFormat != depthImageFrame.Format)
    {
        this.depthImage = null;
        this.depthImageFormat = depthImageFrame.Format;
    }
    if (this.colorImageFormat != colorImageFrame.Format)
    {
        this.colorImage = null;
        this.colorImageFormat = colorImageFrame.Format;
    }

    if (this.depthImage == null)
    {
        this.depthImage = new short[depthImageFrame.PixelDataLength];
    }
    if (this.colorImage == null)
    {
        this.colorImage = new byte[colorImageFrame.PixelDataLength];
    }
    if (this.skeletonData == null || this.skeletonData.Length !=
        skeletonFrame.SkeletonArrayLength)
    {
        this.skeletonData = new Skeleton[skeletonFrame.SkeletonArrayLength];
    }
```

```
        colorImageFrame.CopyPixelDataTo(this.colorImage);
        depthImageFrame.CopyPixelDataTo(this.depthImage);
        skeletonFrame.CopySkeletonDataTo(this.skeletonData);

        foreach (Skeleton skeleton in this.skeletonData)
        {
            if (skeleton.TrackingState == SkeletonTrackingState.Tracked)
            {
                if (skeleton.TrackingId!=trackedID||faceTracker==null)
                {
                    faceTracker = new FaceTracker(kinectSensor);
                    trackedID=skeleton.TrackingId;
                }

                FaceTrackFrame frame = faceTracker.Track(colorImageFormat,
                                                         colorImage,
                                                         depthImageFormat,
                                                         depthImage,
                                                         skeleton);
                Canvas.SetLeft(Face,frame.FaceRect.Left);
                Canvas.SetTop(Face,frame.FaceRect.Top);
                Face.Height = frame.FaceRect.Height;
                Face.Width = frame.FaceRect.Width;
                break;
            }
        }
    }
```

这段代码的主要逻辑以及前面的部分与普通的数据流调用一样。但值得注意的是，Face Tracking不能处理图片格式，遇到这种情况时会抛出异常，因此需要在传入数据前加以判断：

```
    if (this.depthImageFormat != depthImageFrame.Format)
    {
        this.depthImage = null;
        this.depthImageFormat = depthImageFrame.Format;
    }
    if (this.colorImageFormat != colorImageFrame.Format)
    {
        this.colorImage = null;
        this.colorImageFormat = colorImageFrame.Format;
    }
```

另外，最后一段对骨骼数据的循环内容也有所不同。对于每一个追踪到的骨骼，首先要判断其是否为正在跟踪的骨骼，另外，还需判断当前的FaceTracker是否已经初始化，如果不是，就要新建一个FaceTracker取而代之。具体代码如下：

```
    if (skeleton.TrackingId!=trackedID||faceTracker==null)
    {
        faceTracker = new FaceTracker(kinectSensor);
        trackedID=skeleton.TrackingId;
    }
```

然后，调用FaceTracker.Track传入原始数据并获得识别结果。FaceTrackFrame类记录了当前帧的识别结果，其中包含了上一节中提到的全部数据。这里为了便于实现，仅显示了人脸包围盒：

```
FaceTrackFrame frame = faceTracker.Track(colorImageFormat, colorImage,
    depthImageFormat,depthImage, skeleton);
Canvas.SetLeft(Face,frame.FaceRect.Left);
Canvas.SetTop(Face,frame.FaceRect.Top);
Face.Height = frame.FaceRect.Height;
Face.Width = frame.FaceRect.Width;
break;
```

为了配合这段代码，需要在主页面加入相应的Canvas和Rectangle控件，以及用于验证识别结果并显示彩色图的Image控件。主程序的布局文件如下：

```
<Window x:Class="FaceTest.MainWindow"
          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
          Title="MainWindow" Height="480" Width="640" Loaded="Window_Loaded">
    <Grid Name="MainGrid"
          Width="640"
          Height="480">
        <Image Name="ColorImage"/>
        <Canvas Height="480"
                HorizontalAlignment="Left"
                Name="canvas1"
                VerticalAlignment="Top"
                Width="640">
            <Rectangle Canvas.Left="0"
                       Canvas.Top="0"
                       Height="100"
                       Name="Face"
                       Stroke="Yellow"
                       Width="200"
                       StrokeThickness="5" />
        </Canvas>
    </Grid>
</Window>
```

最后，在OnAllFramesReady()函数中加入更新彩色图的语句：

```
this.ColorImage.Source = BitmapSource.Create(colorImageFrame.Width,
                                             colorImageFrame.Height, 96, 96,
                                             PixelFormats.Bgr32, null,
                                             colorImage,
                                             colorImageFrame.Width *
                                             colorImageFrame.BytesPerPixel);
```

至此，这个Face Tracking演示程序就全部完成了，它可以实时地识别出图像中的人脸并用方框圈出，其运行效果如图7-22所示。




图7-22　Face Tracking样例程序运行效果

## 7.5　小结

总体上讲，Kinect for Windows Developer Toolkit中包含的这两个工具都具有极强的实用性。对于Kinect开发者而言，Kinect Studio无疑是一个革命性的里程碑。这种记录和回放功能将Kinect应用的测试流程标准化，通过将一段测试动作记录下来并生成测试用例，就可以批量化地进行Kinect应用测试了。同时，程序员们也摆脱了“一人调试，一人操作”的调试模式。从此，Kinect应用开发进入了标准化、便捷化的时代。

另一方面，新增的Face Tracking SDK弥补了之前SDK对人体头部识别的细节缺失，可以准确地识别出人脸的特征点位置并组成人脸模型，同时还提供了Candide3模型的参数，让开发者可以用最简单的代码开发出功能丰富的人脸应用。相信Face Tracking的脸部识别功能，将会掀起又一波Kinect开发热潮，可以期待不久后即将涌现出一批更加优秀的应用。

第 8 章

# Kinect常用类库介绍

8

在微软的开源工程网站CodePlex（http://www.codeplex.com）上可以找到一些专门为Kinect for Windows SDK编写的扩展类库。目前比较成熟的有两个：一个是Coding4Fun Kinect Toolkit（http://c4fkinect.codeplex.com）；另一个是Kinect Toolbox（http://kinecttoolbox.codeplex.com）。使用这些类库提供的拓展方法可以大幅提高开发效率，省去很多开发成本，因此很有必要在这里向大家介绍一下。

## 8.1　Coding4Fun Kinect Toolkit 介绍

Coding4Fun Kinect Toolkit开源类库为Kinect for Windows SDK提供了一系列的扩展方法，既有基于WPF框架的，也有基于WinForm框架的，本书将着重介绍基于WPF框架的扩展。在官网下载Coding4Fun Kinect Toolkit，然后用添加引用的方法将“Coding4Fun.Kinect.Wpf.dll”添加到工程中，最后在代码头中添加对命名空间的引用，这样就可以调用库中的扩展方法了。

### 8.1.1　基于图像数据的扩展方法



目前，基于图像数据的扩展方法如表8-1所示。

表8-1　Coding4Fun图像数据扩展方法

| 函　　数 | 功能描述 |
| --- | --- |
| ImageFrame.ToBitmapSource() returns BitmapSource | ColorImageFrame类的扩展方法，将ColorImageFrame或DepthImageFrame对象转化为BitmapSource对象 |
| int[].ToBitmapSource() returns BitmapSource | 用于将深度数据数组转化为BitmapSource类型 |
| int[].ToBitmapSource(int width, int height, int minimumDistance, Color highlightColor) returns BitmapSource | 通过设置宽、高、最小距离和颜色，将深度数据数组转化为BitmapSource类型 |

（续）

| 函　　数 | 功能描述 |
|---|---|
| ImageFrame.ToDepthArray()<br>returns int[] | 将DepthImageFrame类型的数据转化为整型数组数据 |
| int[].GetMidpoint(int startX, int startY, int endX, int endY, int minimumDistance)<br>returns Point | 获取深度图像中制定区域的中心点坐标 |
| BitmapSource.Save(string fileName, ImageFormat format)<br>returns nothing | 保存图像信息 |

使用上述方法，就可以将实例1中的图像处理代码改写为：

```
private void kinectSensor_ColorFrameReady(object sender,
    ColorImageFrameReadyEventArgs e)
{
    using (ColorImageFrame imageFrame = e.OpenColorImageFrame())
    {
        if (imageFrame != null)
        {
            //设置图片
            this.ColorImage.Source = imageFrame.ToBitmapSource();
        }
    }
}
```

可以看到，使用了Coding4Fun的函数，将原本烦琐的图像设置缩减到了一行简单的函数调用语句。

另外，使用Coding4Fun Kinect Toolkit库中的扩展方法，还可以将实例2中的深度图像处理代码改写为：

```
const int minDistance = 10;
private void kinectSensor_DepthFrameReady(object sender,
    DepthImageFrameReadyEventArgs e)
{
    using (DepthImageFrame depthImageFrame = e.OpenDepthImageFrame())
    {
        if (depthImageFrame != null)
        {
            //将深度图数据复制到数组中
            pixelData = new int[depthImageFrame.PixelDataLength];
            pixelData = depthImageFrame.ToDepthArray();

            //设置带最小深度限制的深度图
            this.DepthImage.Source = pixelData.ToBitmapSource(depthImageFrame.Width,
                depthImageFrame.Height, minDistance, Colors.Yellow);
        }
    }
}
```

同彩色图一样，Coding4Fun函数也简化了深度图的设置语句，同时还附带了最小深度值限制的功能，避免出现因为距离太近或者太远而导致深度图空白情形。

### 8.1.2 基于骨骼数据的扩展方法

Coding4Fun也提供了骨骼数据的扩展操作，在之前5.4节的实例3中，我们已经使用过了ScaleTo()函数，其详细说明如表8-2所示。

表8-2 Coding4Fun基于骨骼数据的扩展方法

| 函 数 | 功能描述 |
|---|---|
| Joint.ScaleTo(int width, int height) | 将原始骨骼数据中X、Y的值从(–1, 1)转化为(0, width)和(0, height) |
| Joint.ScaleTo(int width, int height, float maxSkeletonX, float maxSkeletonY) | 功能同上一函数，其中maxSkeletonX和maxSkeletonY分别代表原始数据中X、Y的最大值 |

经过这几个例子我们可以看到，利用Coding4Fun Kinect Toolkit库中提供的函数，我们可以简化一些常用的代码，把原来需要的数行代码缩减到一行，这极大提高了应用开发的效率。相信以后该类库还会添加更多简化开发的功能，供开发者使用。

## 8.2 Kinect Toolbox 类库

虽然Kinect SDK能够识别出人体骨骼点的位置，但要将其转化成对应的姿态以及手势仍然需要进行一些识别计算。大多数Kinect应用都需要完成识别过程，因此这一部分也成为Kinect开发过程中重复频率最高的部分。于是，设计一个松耦合、功能完全、定制性高且可重用的识别框架，对众多Kinect开发者是十分必要的，这也是大家绞尽脑汁想要做的事情。在CodePlex网站上，很多开发者都发布了自己的识别框架，其中应用最为广泛的是Kinect Toolbox类库。本节将介绍这个类库的主要功能。



### 8.2.1 Kinect Toolbox 简介

Kinect Toolbox项目主页上把自己定位为一个基于Kinect For Windows SDK的辅助开发工具集，其最主要的功能是为Kinect开发提供一些常用的识别过程，比如人体静态姿态识别、定点轨迹识别（手势识别）、语音识别以及辅助调试等。其中的每种识别都可以分为算法识别和模板识别，即通过识别函数和模板数据加以判别。Kinect的入门开发者可以使用这个类库跳过识别过程，将重点放在其他核心功能上，从而节省开发成本。

同时它提供的辅助调试功能也极为强大，可以直接在WPF画布上绘制骨骼图、骨骼帧记录，调整俯、仰角等。

本节将分步介绍如何在程序中调用Toolbox完成识别功能，并处理其返回事件以及增加自定义姿态。相关源代码及讨论可以查看此项目主页http://kinecttoolbox.codeplex.com。




图8-1　Kinect Toolbox运行示意图

## 8.2.2　人体姿态识别

Toolbox提供的最为稳定的识别方式是骨骼静态姿态识别，通过固定函数判断和模板匹配的方法对每一帧的骨骼点静态姿势进行判断，比如立正、手臂侧平举等。接下来我们将练习在一个新建工程中调用此功能。

### 1. 创建测试窗体并添加引用

首先要创建一个新的WPF项目，如图8-2所示。




图8-2　在Visual Studio 2010中新建测试项目

读者可以在Toolbox官网下载最新的源代码（在http://kinecttoolbox.codeplex.com/SourceControl/list/changesets页面有其版本更新列表），并在解决方案中加入这一项目（使用右键添加现有项目，选择Kinect.Toolbox.csproj），如图8-3所示。

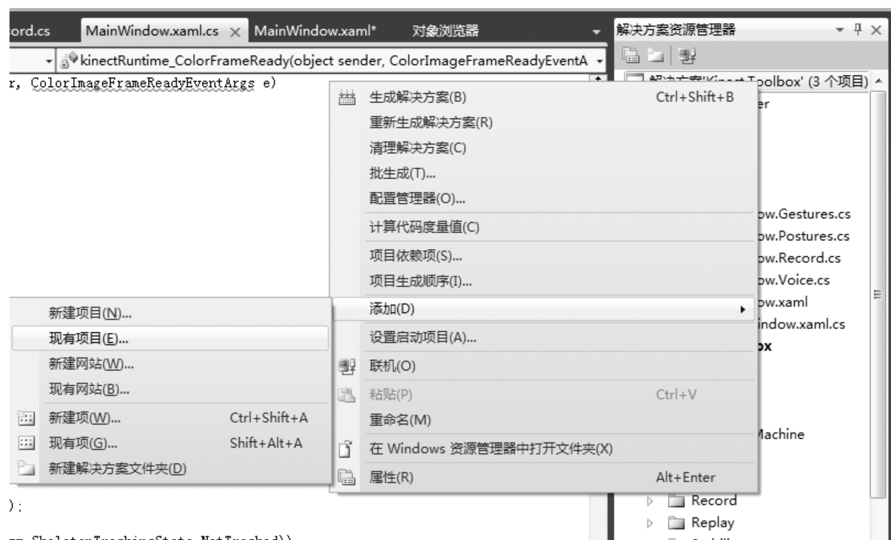



图8-3 添加Toolbox项目

然后在主项目中添加对Toolbox的引用，单击右键添加，选择Kinect Toolbox，如图8-4所示。

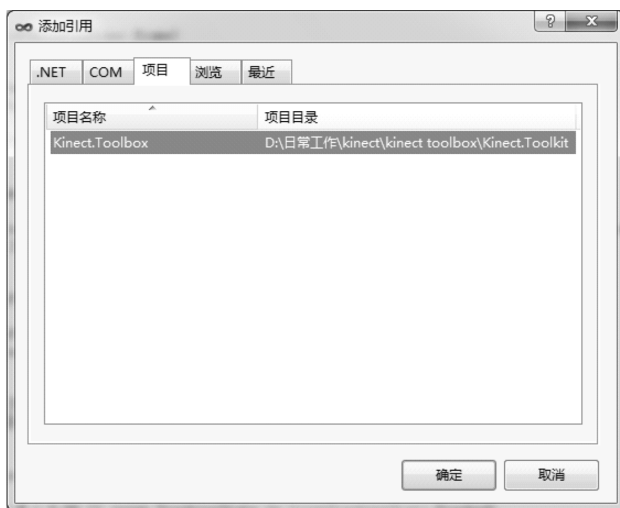



图8-4 添加对Toolbox的引用

在接下来的几节中，我们都将利用这个WPF窗体来完成Toolbox类库的测试以及实验。

### 2. Kinect初始化

在调用Kinect Toolbox进行识别之前，应该先完成Kinect设备的初始化，这在之前的章节中已经做过讲解，这里只将代码列出：

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    try
    {
        KinectSensor.KinectSensors.StatusChanged += Kinects_StatusChanged;

        foreach (KinectSensor kinect in KinectSensor.KinectSensors)
        {
            if (kinect.Status == KinectStatus.Connected)
            {
                kinectSensor = kinect;
                break;
            }
        }

        if (KinectSensor.KinectSensors.Count == 0)
            MessageBox.Show("No Kinect found");
        else
            Initialize();

    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

private void Initialize()
{
    if (kinectSensor == null)
        return;

    kinectSensor.SkeletonStream.Enable(new TransformSmoothParameters{
                                            Smoothing = 0.5f,
                                            Correction = 0.5f,
                                            Prediction = 0.5f,
                                            JitterRadius = 0.05f,
                                            MaxDeviationRadius = 0.04f
                                        });
    kinectSensor.SkeletonFrameReady += kinectRuntime_SkeletonFrameReady;

    kinectSensor.Start();
}
```

### 3. 初始化相应类库

Toolbox的每个功能都被整合在独立的类中，使用时分别调用即可。其中，静态姿态识别过程需要在每一帧骨骼图回调中进行处理。同时，还需使用一个辅助类BarycenterHelper来判断骨骼的稳定状态，所以要先加入这两个类的声明和初始化：

```
readonly BarycenterHelper barycenterHelper = new BarycenterHelper();
readonly AlgorithmicPostureDetector algorithmicPostureRecognizer =
                                        new AlgorithmicPostureDetector();
```

#### 4. 将骨骼数据传入Toolbox并获取识别结果

接下来，需要完成核心的骨骼回调函数，其具体代码如下：

```
void kinectRuntime_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
{
    using (SkeletonFrame frame = e.OpenSkeletonFrame())
    {
        if (frame == null)
            return;

        Tools.GetSkeletons(frame, ref skeletons);
        if (skeletons.All(s => s.TrackingState == SkeletonTrackingState.NotTracked))
            return;

        ProcessFrame(frame);
    }
}

void ProcessFrame(ReplaySkeletonFrame frame)
{
    foreach (var skeleton in frame.Skeletons)
    {
        if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
            continue;

        barycenterHelper.Add(skeleton.Position.ToVector3(), skeleton.TrackingId);
        if (!barycenterHelper.IsStable(skeleton.TrackingId))
            return;

        foreach (Joint joint in skeleton.Joints)
        {
            if (joint.TrackingState != JointTrackingState.Tracked)
                continue;
        }

        algorithmicPostureRecognizer.TrackPostures(skeleton);
        templatePostureDetector.TrackPostures(skeleton);
    }

    currentPosture.Text =
        "Current posture: " + algorithmicPostureRecognizer.CurrentPosture;
}
```

因为Toolbox只能对合法的骨骼数据进行识别，所以要在传入前进行骨骼数据的合法性判断。同时，为了消除抖动对识别的影响，还要将不稳定的骨骼数据移除。具体的方法如下。

(1) 判断骨骼数据的合法性

在之前的章节中已经讲解过`kinectRuntime_SkeletonFrameReady`函数，每当Kinect SDK得到一帧骨骼数据的时候，就会调用这个函数进行处理。因此，我们应先确保这一帧数据不是空

帧，如果是空帧则不应该进行下一步处理：

```
if (frame == null)
    return;
```

如果这一帧的数据不为空，那就要检查其中是否有跟踪到的骨骼，如果没有，也应该返回：

```
Tools.GetSkeletons(frame, ref skeletons);
if (skeletons.All(s => s.TrackingState == SkeletonTrackingState.NotTracked))
    return;
```

这里的Tools是Toolbox提供的一个工具类，GetSkeletons可以从一帧骨骼数据中提取全部的骨骼，并将它们放入一个数组。如果能通过以上两步检验，则表明该帧数据中存在需要分析的骨骼，程序进入ProcessFrame方法。

(2) 判断骨骼是否稳定

骨骼稳定性的判断是由barycenterHelper类来完成的。处理时，应先传入数据，再判断其稳定性，如果不稳定，则不应该继续处理：

```
barycenterHelper.Add(skeleton.Position.ToVector3(), skeleton.TrackingId);
if (!barycenterHelper.IsStable(skeleton.TrackingId))
    return;
```

需要注意的是，在传入参数时，要将骨骼的TrackingId一并传入。

(3) 将骨骼数据传入姿态识别框架

同样地，姿态识别也要将整体骨骼数据传入姿态识别框架：

```
algorithmicPostureRecognizer.TrackPostures(skeleton);
templatePostureDetector.TrackPostures(skeleton);
```

但不同的是，姿态的识别结果并不是以事件回调的方式返回，而是通过一个异步的状态来表示当前的人体姿态，我们可以直接调用这个方法来获取结果：

```
currentPosture.Text =
    "Current posture: " + algorithmicPostureRecognizer.CurrentPosture;
```

最后，需要在wpf窗体上新建一个名为currentPosture的Label的控件来显示识别结果：

```
<Label Content=" "
       Height="28"
       HorizontalAlignment="Left"
       Margin="118,12,0,0"
       Name="currentPosture"
       VerticalAlignment="Top" />
```

运行程序，当左手举过头顶时，系统会识别出此时的人体姿态为LeftHandOverHead。至此，我们就完成了静态姿态类库的全部识别过程。

### 8.2.3　手势识别

与静态姿态识别不同，手势识别表示的是连续的动作，因此其识别算法也要针对连续帧的骨骼数据进行处理。但它们之间也有一个相同点，即手势识别也分为算法判断和模板匹配两种方法。

下面以算法判断为例，识别向左挥手和向右挥手的手势。本节的代码是在上一节代码的基础上添加的。

### 1. 初始化相应类库

与静态识别类似，手势识别的算法被封装在SwipeGestureDetector类中，使用前需要先对其进行声明。

```
SwipeGestureDetector swipeGestureRecognizer;
```

然后，在Initialize()函数中加入其初始化过程，因为需要以事件回调的方式返回识别结果，所以应该加入回调函数注册：

```
swipeGestureRecognizer = new SwipeGestureDetector();
swipeGestureRecognizer.OnGestureDetected += OnGestureDetected;
```

### 2. 传入数据并获取识别结果

手势识别同样需要传入相应的骨骼数据，我们将上一节ProcessFrame的Joint循环的代码修改如下：

```
foreach (Joint joint in skeleton.Joints)
{
    if (joint.TrackingState != JointTrackingState.Tracked)
        continue;
    if (joint.JointType == JointType.HandRight)
    {
        swipeGestureRecognizer.Add(joint.Position, kinectSensor);
    }
}
```

因为仅需要识别右手的挥动状态，所以将右手的骨骼点坐标传入识别库即可。然后，需要完成其回调函数处理返回的识别结果：

```
void OnGestureDetected(string gesture)
{
    switch (gesture)
    {
        /*对不同的手势做相应的处理*/
    }
    currentGesture.Text = "Current gesture: " + gesture;
}
```

其中，gesture记录了识别的结果。读者可能会觉得很困惑，同样是返回识别结果，为什么有的识别是通过回调方法来返回，有的则可以直接通过数据成员获得？这是因为，人体姿态的识别是基于单帧静态骨骼点位置的，其结果只能代表某一帧的骨骼点状态，所以姿态的识别结果就应该在每帧中都进行记录，可以通过一个接口来获取这个结果。而手势识别则不同，它是通过一段时间内右手骨骼点的运动轨迹来进行判别的，因此其识别结果的含义为“之前一段时间内做了某一手势”，这是一个需要响应的即时事件，所以通过回调方法来处理。

最后，仍然要在wpf窗体上新建一个名为currentGesture的Label控件来显示识别结果：

```
<Label Content=" " Height="28"
       HorizontalAlignment="Left"
       Margin="118,50,0,0"
       Name="currentGesture"
       VerticalAlignment="Top" />
```

运行程序，向右挥动右手时，系统可以识别出SwipeToRight手势。至此，我们就完成了全部的手势类库的识别过程。

### 8.2.4　模板识别

前面提到的两种算法识别都是通过固定的函数对单帧或者多帧的数据进行硬性判断，符合条件的才会被识别为对应的姿态和手势。这种方法实现简单，但是缺乏适应性，比如LeftHandOverHead这一姿态，只有在手骨骼点Z轴坐标与头骨骼点Z轴坐标之差大于固定的gap值时才能被识别，而人的体型不尽相同，对于一些臂短的人，即便手举过头顶，可能仍无法满足这一条件。这就需要我们用另一种方法来进行姿态和手势识别，即模板匹配。

Toolbox提供了这两种识别方式对应的模板匹配类库，以及圆形手势和T型姿态的模板用于测试，下面我们将实现这两种算法，仍然在上一节代码的基础上进行修改。

**1. 初始化模板库**

首先，在程序开头添加所需类库的声明：

```
TemplatedPostureDetector templatePostureDetector;
TemplatedGestureDetector circleGestureRecognizer;
```

其中，templatePostureDetector为T型姿态识别类，circleGestureRecognizer为圆形手势识别类。

然后，在初始化函数中添加它们的初始化过程：

```
circleKBPath = Path.Combine(Environment.CurrentDirectory, @"data\circleKB.save");
letterT_KBPath = Path.Combine(Environment.CurrentDirectory, @"data\t_KB.save");

using (Stream recordStream = File.Open(circleKBPath, FileMode.OpenOrCreate))
{
    circleGestureRecognizer = new TemplatedGestureDetector("Circle", recordStream);
    circleGestureRecognizer.OnGestureDetected += OnGestureDetected;
}
templates.ItemsSource = circleGestureRecognizer.LearningMachine.Paths;

using (Stream recordStream = File.Open(letterT_KBPath, FileMode.OpenOrCreate))
{
    templatePostureDetector = new TemplatedPostureDetector("T", recordStream);
    templatePostureDetector.PostureDetected += templatePostureDetector_PostureDetected;
}
postures.ItemsSource = templatePostureDetector.LearningMachine.Paths;
```

由于Toolbox将每一个模板都存储到了文件中，因此需要先读取相应的模板文件，并将其关联到对应的类库中：

```
circleKBPath = Path.Combine(Environment.CurrentDirectory, @"data\circleKB.save");
using (Stream recordStream = File.Open(circleKBPath, FileMode.OpenOrCreate))
{
    circleGestureRecognizer = new TemplatedGestureDetector("Circle", recordStream);
}

using (Stream recordStream = File.Open(letterT_KBPath, FileMode.OpenOrCreate))
{
    templatePostureDetector = new TemplatedPostureDetector("T", recordStream);
}
```

同时，这两个类库也是以时间回调机制来返回识别结果的，因此需要先注册两个回调函数：

```
circleGestureRecognizer.OnGestureDetected += OnGestureDetected;
templatePostureDetector.PostureDetected += templatePostureDetector_PostureDetected;
```

另外需要注意的是，手势和姿态的模板识别都需要机器学习库，Toolbox提供了一个默认的学习库以方便调用。我们仅仅需要设置对应的参数即可。

```
templates.ItemsSource = circleGestureRecognizer.LearningMachine.Paths;
postures.ItemsSource = templatePostureDetector.LearningMachine.Paths;
```

#### 2. 传入数据并获取识别结果

与前两节提到的传入数据方法类似，需要对ProcessFrame函数进行修改，代码如下：

```
void ProcessFrame(ReplaySkeletonFrame frame)
{
    foreach (var skeleton in frame.Skeletons)
    {
        if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
            continue;

        barycenterHelper.Add(skeleton.Position.ToVector3(), skeleton.TrackingId);
        if (!barycenterHelper.IsStable(skeleton.TrackingId))
            return;

        foreach (Joint joint in skeleton.Joints)
        {
            if (joint.TrackingState != JointTrackingState.Tracked)
                continue;

            if (joint.JointType == JointType.HandRight)
            {
                circleGestureRecognizer.Add(joint.Position, kinectSensor);
            }
        }

        templatePostureDetector.TrackPostures(skeleton);
    }
}
```

在上述代码中，原有的静态姿态识别以及手势识别类库被更换为对应的模板类库。

接下来，完成其回调函数以处理识别结果：

```
void teplatePostureDetector_PostureDetected(string posture)
{
    MessageBox.Show("Give me a......." + posture);
}

void OnGestureDetected(string gesture)
{
    MessageBox.Show("Give me a......." + gesture);
}
```

其中，posture和gesture表示了识别的结果。至此，我们就完成了全部的模板类库的识别过程。

### 8.2.5　语音识别

尽管Kinect提供了麦克风阵列，前面的章节也讲解了在SDK中调用Speech API进行语音识别的方法。但在实现这一功能时仍会用到一些重复性的代码，本节将利用Toolbox类库来简化这一过程。

#### 1. 初始化语音库

语音识别库的初始化与Toolbox的其他组件类似：

```
VoiceCommander voiceCommander;
voiceCommander = new VoiceCommander("record", "stop");
voiceCommander.OrderDetected += voiceCommander_OrderDetected;
```

可以看到，这个语音识别库的使用方法非常简便，VoiceCommander类的初始化参数就是需要进行识别的“字典”。而后面则只需添加结果回调方法的绑定。

#### 2. 识别结果回调方法

和手势识别的结果回调类似，语音识别结果的回调也使用了switch语句，对不同的结果做出不同的响应。

```
 void voiceCommander_OrderDetected(string order)
    {
        Dispatcher.Invoke(new Action(() =>
        {
            switch (order)
            {对应不同的识别结果进行相应的处理}
        }));
    }
```

### 8.2.6　添加自定义姿态

在运行之前完成的代码时，我们会发现当前程序只能识别出Toolbox预先编写的几个姿势和手势，这样肯定无法满足不同的应用需求。因此在本节中，我们将尝试在原有框架的基础上添加一个新的姿态识别。

以通过固定算法识别人体姿态为例，首先打开原有框架中的Postures\AlgorithmicPostureDetector.cs

文件，其代码如下：

```
using System;
using Kinect.Toolbox.Record;
using Microsoft.Research.Kinect.Nui;

namespace Kinect.Toolbox
{
    public class AlgorithmicPostureDetector : PostureDetector
    {
        const float Epsilon = 0.1f;
        const float MaxRange = 0.25f;
        public AlgorithmicPostureDetector() : base(10)
        {
        }
        public override void TrackPostures(ReplaySkeletonData skeleton)
        {
            if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
                return;
            Vector3? headPosition = null;
            Vector3? leftHandPosition = null;
            Vector3? rightHandPosition = null;
            foreach (Joint joint in skeleton.Joints)
            {
                if (joint.Position.W < 0.8f ||
                    joint.TrackingState != JointTrackingState.Tracked)
                    continue;

                switch (joint.ID)
                {
                    case JointID.Head:
                        headPosition = joint.Position.ToVector3();
                        break;
                    case JointID.HandLeft:
                        leftHandPosition = joint.Position.ToVector3();
                        break;
                    case JointID.HandRight:
                        rightHandPosition = joint.Position.ToVector3();
                        break;
                }
            }

            // HandsJoined
            if (CheckHandsJoined(rightHandPosition, leftHandPosition))
                return;

            // LeftHandOverHead
            if (CheckHandOverHead(headPosition, leftHandPosition))
            {
                RaisePostureDetected("LeftHandOverHead");
                return;
            }

            {逐个判断各个姿态}
```

8

```
            Reset();
        }

        bool CheckHandOverHead(Vector3? headPosition, Vector3? handPosition)
        {
            if (!handPosition.HasValue || !headPosition.HasValue)
                return false;

            if (handPosition.Value.Y < headPosition.Value.Y)
                return false;

            if (Math.Abs(handPosition.Value.X - headPosition.Value.X) > MaxRange)
                return false;

            if (Math.Abs(handPosition.Value.Z - headPosition.Value.Z) > MaxRange)
                return false;

            return true;
        }

        {其他姿态判别函数}
    }
}
```

其中，`AlgorithmicPostureDetector`类就是Toolbox中负责以固定算法判别姿态的框架。它的识别过程很简单，首先提取姿态判别所需的骨骼点坐标，然后按顺序对每一个姿态进行识别，如果识别结果为真，即当前骨骼数据符合该姿态的要求，那么就认为该姿态为当前人体姿态，结束识别过程。我们的自定义姿态——向前伸出手臂，也将按照这种方式添加。

向前伸出手臂的判定条件很简单，如果右肩骨骼点的Z坐标与右手骨骼点的Z坐标之差大于固定阈值，即可认定右手已经向前伸出；左手同理。因此相比原有框架，需要再多记录两个肩膀的骨骼点坐标：

```
case JointID.ShoulderLeft:
    leftShoulder = joint.Position.ToVector3();
    break;
case JointID.ShoulderRight:
    rightShoulder = joint.Position.ToVector3();
    break;
```

与LeftHandOverHead的判定结果类似，将两个伸出手臂的判定添加到顺序识别过程中。这里要注意两只手同时伸出的情况：

```
// HandsReachout
bool leftHandReachout = CheckHandReachout(leftHandPosition, leftShoulder);
bool rightHandReachout = CheckHandReachout(rightHandPosition, rightShoulder);
if (leftHandReachout&&rightHandReachout)
{
    RaisePostureDetected("BothHandReachout");
    return;
}
else if (leftHandReachout)
```

```
{
    RaisePostureDetected("LeftHandReachout");
    return;
}
else if (rightHandReachout)
{
    RaisePostureDetected("RightHandReachout");
    return;
}
```

最后，我们要完成CheckHandReachout函数，使用该函数比较手和肩膀骨骼点的位置并得到姿态识别结果：

```
bool CheckHandReachout(Vector3? headPosition, Vector3? handPosition)
{
    if (!handPosition.HasValue || !headPosition.HasValue)
        return false;
    if (handPosition.Value.Z - headPosition.Value.Z < ReachoutRange)
        return false;
    return true;
}
```

需要注意的是，在整个识别过程中的骨骼点坐标数据使用的都是Vector3?数据类型。因为在骨骼数据中有可能存在少量未被识别的骨骼点，此时是不能进行姿态识别的，所以要先进行合法性判断。

到目前为止，我们已经将手势、姿态以及语音识别的功能全部实现了，并且依靠Toolbox类库，节省了很多重复性的框架代码。当然，它并不是唯一的识别库，其他很多开源识别库也有各自的优点，这里仅仅是用它做一个例子，帮助读者理解手势及姿态的识别过程。其他识别库的使用方法大同小异，有兴趣的读者可以对其进行更加深入的研究，这里就不作过多的介绍了。

## 8.3 小结



Coding4Fun Kinect Toolkit和Kinect Toolbox这类第三方类库从功能上看和第7章中提到的Kinect Studio和Face Tracking SDK类似，都在很大程度上简化了代码的复杂程度并且大大提高了开发效率。区别在于后者是由微软官方提供的类库，相比前者有更好的稳定性以及更完整的文档说明。而使用第三方类库时最常见的一个问题就是文档不完整，甚至没有文档，开发者仅能通过阅读源代码或者示例代码来熟悉使用方法，较为麻烦。另外，有可能在版本更迭的时候由于一些没有记入文档的改动而出现莫名其妙的问题，读者在使用的时候应该加以注意。

类似的第三方类库还有很多，这里就不一一列举了，读者可以到各大开源社区查看相应的项目主页自行学习。

*Part 3*

第三部分

# Kinect 实战篇

到现在为止，我们已经掌握了 Kinect for Windows SDK 的 API 基本调用方法，但距离开发 Kinect 应用还有一段路要走。在开发 Kinect 应用时，仅凭正确地调用 API 参数是不够的，作为实际的应用程序，其应用领域、优劣势以及用户体验都需要开发者仔细地思考。同时，由于 Kinect 应用开发尚处于起步阶段，常用功能的类库还比较少，很多基础功能的代码都需要开发者自行编写。

因此，为了让读者能够尽快入门 Kinect 应用开发，本书将在实战篇中结合大量的实例项目，从教育辅助、文化保护和机械控制等应用方面介绍 Kinect for Windows 的实用开发技巧。其中包含了 Kinect 开发中的常见问题，而且应用背景和项目切入点也具有相当高的参考价值。

# 第9章 Kinect虚拟演示系统的实现

如何跳出动作识别这一思维定势，并利用Kinect实现更有创意的功能？如何设计出优秀的交互方式，既能让用户使用自如，又能最大限度地降低误识别率？在进行Kinect应用构思时，我们经常会遇到这些问题，本章介绍的项目就很好地回答了这两个问题。

该项目是在2011年微软亚洲研究院举办的“Kinect Pioneer”大赛中获得第一名的作品，它不仅巧妙地使用Kinect SDK提供的数据完成了平时较难解决的人体抠图问题，而且在演示具体的使用场景时还进行了创新，设计出一种新颖的演示模式。同时，其易于使用的整体自然交互设计也是值得Kinect开发者学习和研究的。

## 9.1 虚拟演示系统简介

当下，“演示”已经成为我们生活中不可或缺的一部分，比如课堂演示、演讲展示以及商业展示等。所有这些以多媒体手段作为辅助语言来进行表达的行为，都可以称为演示。我们平时看到的演示活动大多使用幻灯片作为内容的载体，利用大屏幕或者投影仪将内容显示出来，而演示人会在演示的同时利用鼠标等控制器来进行幻灯片的切换。因此，利用PowerPoint软件制作的PPT演示文稿，凭借其简单的制作过程和强大的功能，几乎已经成为了演示的代名词。

PPT演示方法具备很多优势，但在一些情况下也存在缺点。首先，演讲人必须依靠鼠标或者其他控制器才能完成幻灯片的切换，频繁的鼠标操作会导致演示行为极不自然；其次，在一些特殊的演示场景中，比如远程课堂、视频会议等，演示行为仅能通过一个屏幕来显示，如何兼顾演讲者图像以及幻灯片图像就成了一个问题。前面提到的虚拟演示系统就是为了解决这些问题而出现的一种全新的演示方案，如图9-1所示。演示者的图像嵌入到了场景中，同时演示者可以与场景中的元素互动，比如抓取场景中的星球，拖拽并且放大等。

为了避免频繁的鼠标操作，该系统使用自然交互方式代替了传统交互方式。演讲者可以通过手势、姿态以及语言来进行演示内容的切换和交互，这使得幻灯片切换可以和演讲自然地结合在一起。

此外，兼顾演讲者以及幻灯片图像最好的方式就是将人物图像抠取出来，并放入演示内容当中。这样不仅能实现“一屏多用”，而且还可以表现出演讲者在虚拟环境中演讲的效果（类似于虚拟演播室的效果），从而提高观众的观看体验。

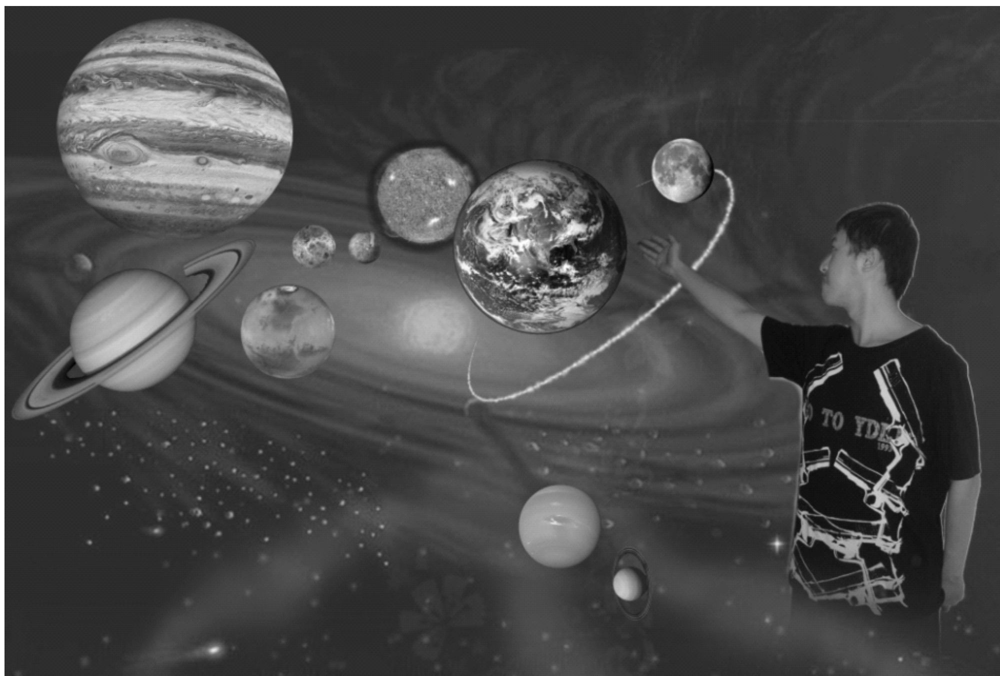


图9-1 虚拟环境演示系统

## 9.2 技术实现概述

我们可以利用Kinect提供的数据，来实现系统中最主要的两个需求——人物图像抠取和自然交互控制演示内容。其中，Kinect SDK提供的深度数据流在记录每个像素深度数据的同时，也标记该像素属于哪一个用户，并且这种标记是连贯性的，可以持续跟踪用户。同时，还可以利用`MapDepthToColorImagePoint`函数将该标记对应到RGB图像中，这样即可从RGB图像中抠取出人物图像。

而自然交互部分需要对Kinect SDK提供的原始骨骼数据进行手势和姿态识别，并通过Microsoft Speech API进行语音识别，因此，必须设计一套合理的交互方式来控制演示内容。

同时，由于整个系统的演示需求已经超出了传统PPT所能达到的效果（需要实现人和演示内容交互），所以需要使用图像引擎重新实现一个演示界面，既具有幻灯片的效果，又可以实现交互功能。

本章会在后面几节重点分析上面提到的Kinect SDK的两个功能，并列出相应的实现代码，但不会对该项目的整体架构做过多讲解。演示框架部分不是本书的重点，因此仅做简单介绍。另外，该项目代码是基于XNA 4.0框架编写的，对XNA不熟悉的读者可以先阅读9.5节或者其他相关的书籍，初步了解XNA框架的运行方式，便于理解代码。

## 9.3　利用深度数据标签获取人物彩色图像

我们将从这一节开始分析技术实现细节。首先来完成一个人物抠图类，它利用Kinect SDK完成人物彩色图像抠取功能，并对其进行修补和优化。该类会在初始化时传入Kinect对象，并在XNA轮询框架的Update过程中调用，用于更新人物的彩色图像。另外，在绘制过程中也会被调用，用于绘制当前的人物图像。

这里，假设Kinect的初始化工作已经在其他类中完成，我们仅针对“人物图像抠取”功能进行代码实现。

### 9.3.1　创建人物抠图类

在Visual Studio 2010中新建一个C#类，命名为“Human.cs”，如图9-2所示。我们将在这个类中完成本节的全部工作。

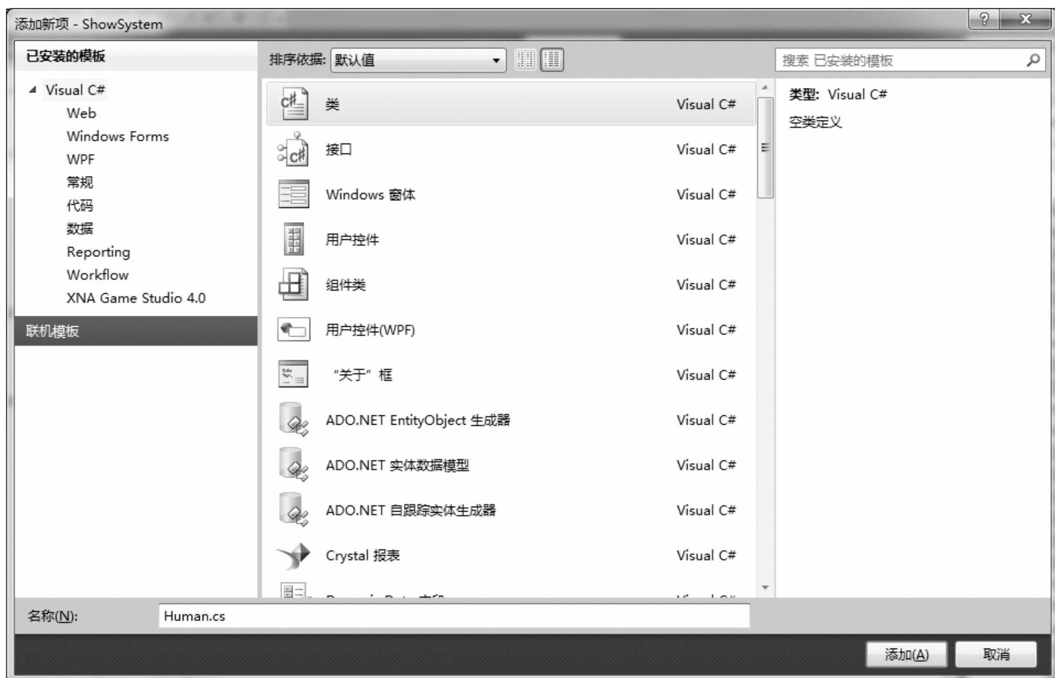



图9-2　在Visual Studio 2010中新建人物抠图类

### 9.3.2　利用深度数据获取人物彩色图像

获取人物彩色图像的大致流程如图9-3所示。

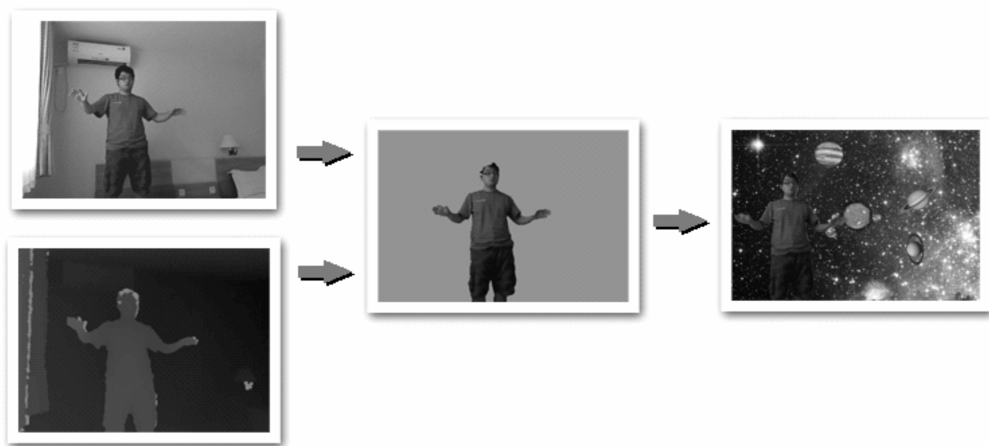


图9-3 人物抠图方法示意图

在深度图中，Kinect SDK可以对每个像素进行用户索引标记，同时提供深度图坐标到平面图坐标的转换函数。这样我们就可以得到某一个用户的彩色像素集，将彩色图中的其他像素都设为透明，即可得到最终只包含某一用户的彩色图像，具体代码如下：

```
private Color[] getBody(KinectSensor nui)
{
    colorImage = nui.ColorStream.OpenNextFrame(1);
    depthImage = nui.DepthStream.OpenNextFrame(1);

    if (colorImage == null || depthImage == null)
        return null;

    depthImage.CopyDepthImagePixelDataTo(depthFrame);
    colorImage.CopyPixelDataTo(colorFrame);

    nui.CoordinateMapper.MapDepthFrameToColorFrame(
                depthImage.Format,
                depthFrame,
                colorImage.Format,
                colorCoordinates);

    bodyOnlyColor = new Color[colorWidth * colorHeight];

    for (int depthY = 0, depthIndex = 0; depthY < depthHeight; depthY++)
    {
        for (int depthX = 0; depthX < depthWidth; depthX++, depthIndex += 1)
        {
            DepthImagePixel depthPixel = depthFrame[depthIndex];

            int player = depthPixel.PlayerIndex;

            if (player != 0 && userID == -1)
                userID = player;
```

9

```
                if (player == userID && player != 0)
                {
                    ColorImagePoint colorPoint = colorCoordinates[depthIndex];

                    int colorX = Math.Max(0, Math.Min(colorPoint.X,
                                 colorImage.Image.Width - 1));
                    int colorY = Math.Max(0, Math.Min(colorPoint.Y,
                                 colorImage.Image.Height - 1));

                    int colorIndex = (colorY * colorWidth + colorX);
                    bodyOnlyColor[colorIndex] = new Color(colorFrame[colorIndex + 2],
                                                          colorFrame[colorIndex + 1],
                                                          colorFrame[colorIndex],
                                                          255);
                }
            }
        }
        return bodyOnlyColor;
    }
```

在上述代码中，我们实现了一个函数，该函数接收`KinectSensor`对象，并返回人物彩色图像的`Color`数组，实现了利用深度图抠取人物彩色图的功能。下面我们将对这些功能进行更深入的分析。

**1. 获取彩色图和深度图数据**

首先利用`ColorStream`和`DepthStream`的接口获取一帧彩色图像和深度图像：

```
colorImage = nui.ColorStream.OpenNextFrame(1);
depthImage = nui.DepthStream.OpenNextFrame(1);
```

值得注意的是，与之前介绍的回调事件不同，这两个接口是同步阻塞的调用函数。这里有必要简单解释一下同步调用和异步调用的区别。它们最主要的不同点在于，同步请求是在发出调用请求后，停止进程的执行，直到获得请求的数据或者达到预定的等待时间，才继续执行之后的处理代码；而异步请求在发出请求后并不阻塞进程的执行，而是在数据返回时捕获其事件并执行相应的回调函数进行处理。这两种数据请求的处理方式各有特点：同步请求的效率高，适合在轮询框架中使用；异步请求的逻辑清晰，适合在事件响应框架中使用。因为我们使用XNA的轮询框架作为程序的主体逻辑框架，所以同步的方法在这里更适合。

在上述代码中，`OpenNextFrame()`方法用于获得彩色流或深度流的下一帧图像，其参数表示该函数等待的毫秒数。这里采用同步阻塞型方法来获取数据是为了提高程序的效率。

因为是同步阻塞型方法，所以有可能在某次调用的时候无法获取数据。这时不应该进行抠图操作，相关代码如下：

```
if (colorImage == null || depthImage == null)
    return null;
```

最后，将两种图像的字节数组分别提取出来，相关代码如下：

```
depthImage.CopyDepthImagePixelDataTo(depthFrame);
colorImage.CopyPixelDataTo(colorFrame);
```

### 2. 深度图像和彩色图像的存储方式

下面我们先来了解深度图像和彩色图像的字节数组存储方式。如果对图像上的每个像素都采用如图9-4所示的方式计数，那么深度图字节数组中每个元素的含义就如表9-1所示。

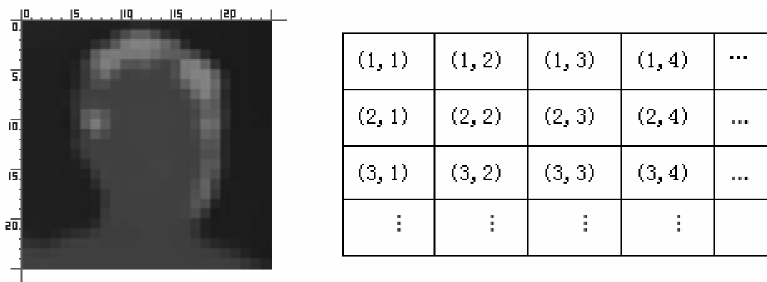



图9-4　图片存储方式

表9-1　深度图数据对应含义

| 数组下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 含义 | (1,1)的深度值和人物标记 | | (1,2)的深度值和人物标记 | | (1,3)的深度值和人物标记 | | (1,4)的深度值和人物标记 | |

其中，两个合并在一起的字节表示对应像素的深度值和人物标记。而彩色图字节数组中每个元素的含义稍有不同，如表9-2所示。

表9-2　彩色图数据对应含义

| 数组下标 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 含义 | (1,1)像素RGBA值，4个字节分别表示B、G、R、A的数值 | | | | (1,2)像素RGBA值，4个字节分别表示B、G、R、A的数值 | | | |

需要注意的是，彩色图是用4个字节来表示一个像素的。

### 3. 利用深度图标记彩色图像素

首先使用循环来遍历深度图上的每一个像素，并获取其深度值和人物标记，这在之前的章节中已经做过介绍。

```
for (int depthY = 0, depthIndex = 0; depthY < depthHeight; depthY++)
{
    for (int depthX = 0; depthX < depthWidth; depthX++, depthIndex += 1)
    {
        DepthImagePixel depthPixel = depthFrame[depthIndex];

        int player = depthPixel.PlayerIndex;
    }
}
```

其中，depthPixel记录了这个深度图像素的全部数据信息，Player表示该像素上的人物标记（关于深度数据的具体数据结构可参考4.1节）。值得一提的是，该人物标记值可以持续地跟踪某一人体，并返回相同的标记值。但是如果该人体离开Kinect视野后再度返回，这一标记值有可能发生改变。换言之，用户离开Kinect视野后，再度返回时可能会被Kinect视为另一个人。

然后利用CoordinateMapper.MapDepthFrameToColorFrame()方法得到对应的彩色图像素坐标：

```
nui.CoordinateMapper.MapDepthFrameToColorFrame(
            depthImage.Format,
            depthFrame,
            colorImage.Format,
            colorCoordinates);
ColorImagePoint colorPoint = colorCoordinates[depthIndex];
```

值得注意的是该方法的第四个参数colorCoordinates，这是这个坐标转换函数的结果数组，可以用colorCoordinates[depthIndex]的方法来获取对应该depthIndex下的彩色图坐标。

如此得到的彩色图坐标可能会超出彩色图的宽度、高度值，因此还要将其转化为合法数据：

```
int colorX = Math.Max(0, Math.Min(colorPoint.X, colorImage.Image.Width - 1));
int colorY = Math.Max(0, Math.Min(colorPoint.Y, colorImage.Image.Height - 1));
```

最后，换算出Color数组的下标，并在相应的位置保存这一像素值：

```
int colorIndex = (colorY * colorWidth + colorX);
bodyOnlyColor[colorIndex] = new Color(colorFrame[colorIndex + 2],
                                      colorFrame[colorIndex + 1],
                                      colorFrame[colorIndex],
                                      255);
```

其中，bodyOnlyColor是Color类型的数组，而Color则是XNA框架定义的颜色类型，可以理解为一个保存单像素RGBA四位数值的对象。

#### 4. 持续跟踪某一固定标记的人物

由于Kinect可以同时识别多个人物，而我们仅希望对某一个人物进行持续跟踪，所以要在第一次出现人物的时候记录下该人物标记，并在之后的每一帧都只对该标记的像素进行处理，具体代码如下：

```
int player = depthPixel.PlayerIndex;
if (player != 0 && userID==-1)
    userID = player;
if (player == userID && player != 0)
{
    // 将深度图像素对应到彩色图
}
```

至此，我们就完成了一个具备最基本功能的抠图方法。如果将此方法替换工程中原有的部分并执行，我们就可以看到抠取出来的人物了，如图9-5所示。



图9-5 人物抠取初步结果

### 9.3.3 修补、优化并完善抠图类

可以看到，现在得到的图片还有很多瑕疵，图像边缘的噪点很多，极其不光滑，人物图中还经常会出现一些小“空洞”。这是因为到目前为止仅仅使用了原始的人物标记数据进行了图像抠取，使得整个图像显得很粗糙。因此需要通过修补和优化来让这幅抠图显得更加自然。我们可以在得到colorX和colorY之后加入如下代码：

```
for (int j = colorY - 3; j <= colorY + 3; j++)
{
    for (int i = colorX - 3; i <= colorX + 3; i++)
        if (j >= 0 && j < colorHeight && i >= 0 && i < colorWidth)
        {
            int t32 = (j * colorWidth + i) * 4;
            int t16 = (j * colorWidth + i);
            pixeltype[i, j] += 1;
            if (pixeltype[i, j] > 7)
            {
                bodyOnlyColor[t16] = new Color(lastColor[t32 + 2],
                                               lastColor[t32 + 1],
                                               lastColor[t32], 255);
            }
        }
}
```

这里使用了一种类似于Photoshop中“羽化”的操作：对于彩色图中的每个像素点，统计以其为中心的方形范围内有多少像素是深度图中对应的像素点。如果超过某一阈值，那么该像素点就可以被认为属于人物标记而加入结果图像。

但是使用这种方法存在一个问题，即在仅仅统计该方形范围内被选取像素点的个数时，系统会将不同位置的像素视为同等价值。然而对于边缘外的点以及内部的“空洞”，尽管覆盖了12个像素，但实际上它们是不能等同的，如图9-6所示，蓝色点代表已经选择的像素点，红色点代表当前需要判断的像素点，白色点代表未被选择的像素点。

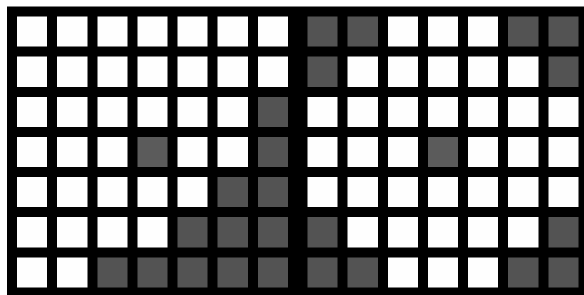


图9-6　不同位置的选取策略

因此我们要对这个方形范围内的每一个位置添加相应的权值：

```
private int[,] pixelNum = {{0, 0, 0, 4, 0, 0, 0},
                           {0, 0, 4, 3, 4, 0, 0},
                           {0, 4, 3, 2, 3, 4, 0},
                           {4, 3, 2, 1, 2, 3, 4},
                           {0, 4, 3, 2, 3, 4, 0},
                           {0, 0, 4, 3, 4, 0, 0},
                           {0, 0, 0, 4, 0, 0, 0}};
```

然后修改统计的方法：

```
pixeltype[i, j] += pixelNum[i - colorX + 3, j - colorY + 3];
if (pixeltype[i, j] > 10)
    {加入该像素}
```

上述代码中的权值仅仅是通过实验得到的一组较为合适的参数，并没有在理论上进行验证，有兴趣的读者可以进行深入的研究。同时，本节介绍的这种优化算法并不是唯一的方法，在数字图像处理技术中还有很多算法可以应用到这里，甚至能得到更理想的结果。但是由于效率以及框架兼容性等问题，我们最终选用了这种轻量级的算法，以平衡抠图的效率和质量。

至此，我们就实现了全部的人物图像抠取功能，并且可以得到一幅较为理想的人物抠图了，如图9-7所示。



图9-7　人物抠取最终结果

### 9.3.4　利用Kinect SDK抠图的优、缺点

其实，还有很多种方法可以实现人物图像抠取这一功能，比如虚拟演播室中采用的单色幕布技术，当用户站在单色幕布前面时，只要去除摄像机实时拍摄画面中的幕布颜色，即可将人物图像完整地抠取出来。此外，还可以使用一些机器视觉算法来识别图像中的人体，并加以抠取。但是与这些较为完善的解决方案相比，利用Kinect进行抠图仍然有很多优点。

首先是稳定性。由于Kinect是使用深度图区分人物和背景的，因此不会受到光线明暗、衣着颜色和背景颜色的影响，甚至在暗室中也可以区分出人物和背景，这是传统的机器视觉算法所不能比拟的。另外，利用Kinect抠图极其便捷，不需要任何辅助设备，只要有一台Kinect和一台电脑即可，相比单色幕布技术优势很大。

当然，这种方法也有一些自身无法克服的缺陷，比如只能依赖Kinect SDK抠取人物图像，而其他方法并不受限于此。同时无法通过硬件提高图像分辨率，无法达到那些高精度、高质量人物抠图工作的要求。但总体来说，这种方法已经在很大程度上超越了传统的方法，相关的应用也会越来越多。

## 9.4　利用骨骼数据识别人体姿态

虚拟演示系统的第二个主要功能是识别人体动作和姿态，我们将在本节中介绍其实现方法以及设计思路。

### 9.4.1　利用Toolbox实现主体识别功能

基本的人体姿态和手势识别功能，可以通过第8章介绍的Toolbox库来实现。Toolbox库的具体功能就不在这里介绍了，本节将重点讨论在实际应用中使用Kinect识别的一些细节问题，比如持续跟踪、交互方式设计等。

1. 添加引用并创建Nui类

首先下载最新版本的Toolbox类库，然后在主项目中添加对Toolbox的引用，可参照8.2.2节的步骤完成配置。接下来，在Visual Studio 2010中新建一个C#类，命名为“Nui.cs”，如图9-8所示。我们将在这个类中完成Toolbox类库所需对象的初始化，并传入数据进行识别。

2. Toolbox初始化

首先完成该类的构造函数，即初始化过程，相关代码如下：

```
SwipeGestureDetector swipeGestureRecognizer;
TemplatedGestureDetector circleGestureRecognizer;
readonly BarycenterHelper barycenterHelper = new BarycenterHelper();
readonly AlgorithmicPostureDetector algorithmicPostureRecognizer =
    new AlgorithmicPostureDetector();
TemplatedPostureDetector templatePostureDetector;

public Nui(KinectSensor nui)
{
```

```
        kinectSensor = nui;
        circleKBPath = Path.Combine(Environment.CurrentDirectory, @"data\circleKB.save");
        letterT_KBPath = Path.Combine(Environment.CurrentDirectory, @"data\t_KB.save");

        try
        {
            swipeGestureRecognizer = new SwipeGestureDetector();
            swipeGestureRecognizer.OnGestureDetected += OnGestureDetected;
            kinectSensor.SkeletonFrameReady += kinectRuntime_SkeletonFrameReady;

            LoadCircleGestureDetector();
            LoadLetterTPostureDetector();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message);
        }
    }

    void LoadCircleGestureDetector()
    {
        using (Stream recordStream = File.Open(circleKBPath, FileMode.OpenOrCreate))
        {
            circleGestureRecognizer =
                new TemplatedGestureDetector("Circle", recordStream);
            circleGestureRecognizer.OnGestureDetected += OnGestureDetected;
        }
        templates.ItemsSource = circleGestureRecognizer.LearningMachine.Paths;
    }

    void LoadLetterTPostureDetector()
    {
        using (Stream recordStream = File.Open(letterT_KBPath, FileMode.OpenOrCreate))
        {
            templatePostureDetector = new TemplatedPostureDetector("T", recordStream);
            templatePostureDetector.PostureDetected +=
                templatePostureDetector_PostureDetected;
        }

        postures.ItemsSource = templatePostureDetector.LearningMachine.Paths;
    }
```

我们在构造函数之前声明了一系列数据成员，这些数据成员就是在Kinect Toolbox中实现的一些辅助类库，可以完成识别、记录等工作。

- swipeGestureRecognizer：固定算法识别挥动手势。
- circleGestureRecognizer：模板匹配识别圆圈手势。
- barycenterHelper：判断当前人物是否静止。
- algorithmicPostureRecognizer：固定算法识别人体姿态。
- templatePostureDetector：模板匹配识别T字姿态。

其中，swipeGestureRecognizer、circleGestureRecognizer和templatePostureDetector是以事件回调方式来返回识别结果的，因此需要借助相应的回调方法来处理事件：

```
swipeGestureRecognizer.OnGestureDetected += OnGestureDetected;
circleGestureRecognizer.OnGestureDetected += OnGestureDetected;
templatePostureDetector.PostureDetected += templatePostureDetector_PostureDetected;
```

打开骨骼数据流的事件回调方法如下：

```
kinectSensor.SkeletonFrameReady += kinectRuntime_SkeletonFrameReady;
```

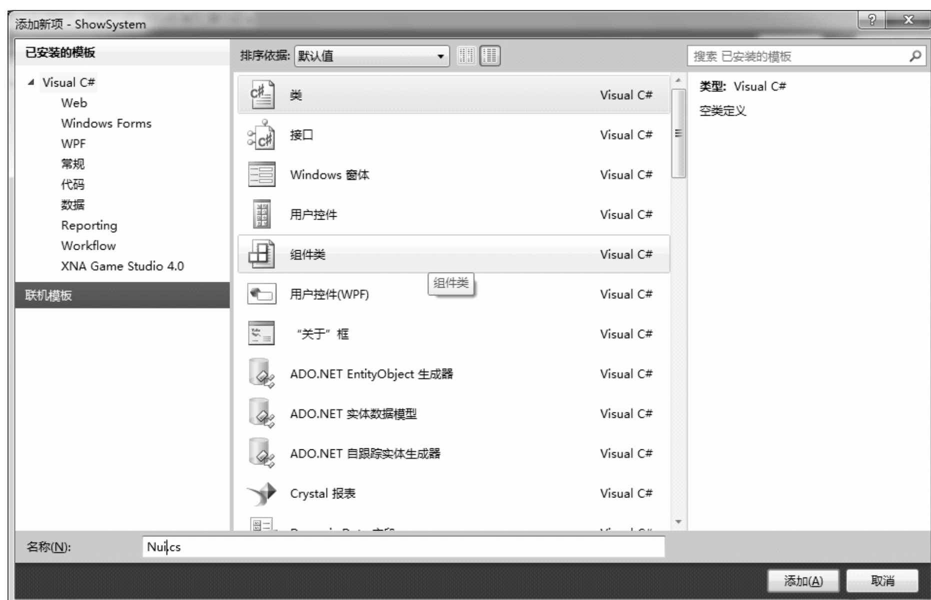



图9-8 在Visual Studio 2010中新建Nui类

### 3. 将骨骼数据传入Toolbox并获取识别结果



按照惯例，我们仍然需要在骨骼数据的回调函数中将Kinect获取的骨骼点信息传入Toolbox，相关代码如下：

```
void kinectRuntime_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
{
    using (SkeletonFrame frame = e.OpenSkeletonFrame())
    {
        if (frame == null)
            return;
        Tools.GetSkeletons(frame, ref skeletons);
        if (skeletons.All(s => s.TrackingState == SkeletonTrackingState.NotTracked))
            return;

        ProcessFrame(frame);
    }
```

```
}

void ProcessFrame(ReplaySkeletonFrame frame)
{
    foreach (var skeleton in frame.Skeletons)
    {
        if (skeleton.TrackingState != SkeletonTrackingState.Tracked)
            continue;

        if (userID == -1)
            userID = skeleton.TrackingID;
        if (skeleton.TrackingID != userID)
            continue;

        barycenterHelper.Add(skeleton.Position.ToVector3(), skeleton.TrackingId);
        if (!barycenterHelper.IsStable(skeleton.TrackingId))
            return;

        foreach (Joint joint in skeleton.Joints)
        {
            if (joint.TrackingState != JointTrackingState.Tracked)
                continue;
            if (joint.JointType == JointType.HandRight)
            {
                swipeGestureRecognizer.Add(joint.Position, kinectSensor);
                circleGestureRecognizer.Add(joint.Position, kinectSensor);
            }
        }
        algorithmicPostureRecognizer.TrackPostures(skeleton);
        templatePostureDetector.TrackPostures(skeleton);
    }
    currentPosture = algorithmicPostureRecognizer.CurrentPosture;
}

void OnGestureDetected(string gesture)
{
    switch (gesture)
    {
        // 对不同的手势做相应的处理
    }
}

void templatePostureDetector_PostureDetected(string posture)
{
    switch (posture)
    {
        // 对不同的姿势做相应的处理
    }
}
```

这里的大部分代码和普通的调用方式相同。但值得注意的是，和深度图的跟踪类似，骨骼数据也应该只对某一特定编号的骨骼进行持续跟踪和识别：

```
if (userID == -1)
    userID = skeleton.TrackingID;
if (skeleton.TrackingID != userID)
    continue;
```

至此，我们已经实现了使用骨骼数据识别人体姿态和手势的基本功能——初始化、传入数据以及获取结果。

### 9.4.2　自然交互方式设计

对于一个完美的Kinect应用来说，识别功能仅仅是基础而已，更重要的是将多种识别组合在一起，设计出一个适合该应用的统一自然交互系统，这将直接关系到应用的质量。试想，如果一个演示应用以手举过头顶的动作来完成翻页，同时还夹杂着手势和语音的误识别，其用户体验一定是非常糟糕的。下面我们就以虚拟演示系统为例，详细讨论如何设计一个完整的自然交互系统。

**1. 各种识别方式的特性**

针对Kinect的自然交互功能，我们已经介绍了三种不同的识别方式：静态姿态识别、动态手势识别以及语音识别。这三种识别方式的优缺点和适用的环境都不尽相同，区别如表9-3所示。

**表9-3　不同识别方式的特性**

| 识别方式 | 识别功能 | 优　点 | 缺　点 |
| --- | --- | --- | --- |
| 姿态识别 | 识别某一时刻人体的静态姿态 | 稳定：由于仅仅需要对单帧的数据进行识别，所以稳定性很高 | 适用范围较窄：只能用于识别一些静态的姿态，比如立正、伸出手臂等。另外，识别的结果为当前帧的“状态”，而不是做出某一动作的“事件” |
| 手势识别 | 识别一小段时间内手的连续动作 | 交互方式自然：手势本身就是连贯的动作，动态识别符合人的思维方式 | 误识别率高：因为需要连续匹配多帧的数据，所以造成了较高的误识别率。<br>识别复杂度低：因为手势越复杂，误识别率就会越高，所以只能识别一些简单的手势 |
| 语音识别 | 识别语音指令 | 交互方式自然：有些操作直接用语音进行驱动，更加直接和自然 | 误识别率高：会将很多杂音识别为语音库的声音 |



对应于这些特性，它们的使用环境也不太一样：姿态识别凭借其出色的稳定性，一般适用于控制交互状态的切换；手势识别利用其与人体动作的高度契合而作为主要的交互驱动方式；而语音识别则对其他两种识别方式的一些盲点操作进行补充，并不作为交互的主体部分。

**2. 应用所需的交互功能**

当然，仅仅了解识别方式的特点显然是不够的，对于不同的应用，我们应该根据其本身的使用特点，设计出最合适的自然交互系统。因此，另一项必须提前完成的功课就是分析应用的自然交互需求。以演示系统为例，表9-4列出了它所需要的交互操作及其特点。

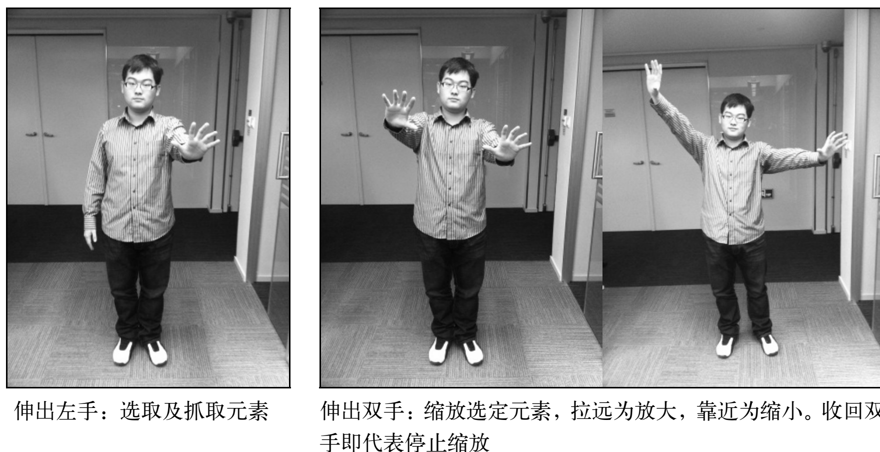
表9-4　不同的交互需求

| 操作类型 | 需求描述 |
| --- | --- |
| 选择元素 | 元素指页面中的图片、文本和视频。需要选中当前页面中的某一元素 |
| 移动元素 | 在页面中移动选中的元素 |
| 放大、缩小元素 | 放大和缩小被选中的元素 |
| 切换元素 | 类似于传统PPT的动画切换效果，对选中的元素进行动画切换 |
| 前后翻页 | 将页面转到上一页或下一页 |
| 进入、退出子页面 | 针对当前页面的某些元素，将页面转到以其为主题的子页面中 |

### 3. 组合出统一的交互系统

从前面的交互需求分析可以看出，演示系统的交互主要分为两种——对元素的操作以及对页面的控制。因此，在交互设计过程中也要将这两类操作区分开来。

首先分析对元素的操作，其中选取、移动操作相当于鼠标的选取和拖动。通常这类操作的实现过程是将手的坐标映射到平面上的鼠标坐标，然后通过延时操作，当鼠标在某一元素上停留一段时间后，就认为这个元素已经被选取了。需要注意的是，为了让用户知道延时的进程，需要将延时过程显示出来。另外，针对放大和缩小操作，最直观的方式是用两只手来模拟多点触控中的放大和缩小，即双手距离拉远为放大，靠近为缩小，如图9-9所示。



伸出左手：选取及抓取元素　　伸出双手：缩放选定元素，拉远为放大，靠近为缩小。收回双手即代表停止缩放

图9-9　元素类操作示意图

不难发现，以上操作均需要通过伸出双手来完成。那么剩下的元素切换操作也可以通过类似的手势识别来完成，比如向左、向右挥动右手，这样全部的元素操作就具有了统一的特性。因此系统可以通过姿态识别出向前伸手的状态，并在此状态下开启元素的操作，在其他状态下则关闭这类操作。这样的设计可以区分两种操作状态，并避免它们相互干扰。

接下来对于页面的控制就相对简单多了，设计一些不用向前伸手的姿态即可，如图9-10所示。



右手置于头右侧：进入、返回子页面　　左臂平举：返回上一页面　　右臂平举：进入下一页面

图9-10　控制类操作示意图

这样，我们就完成了演示系统全部的自然交互设计。整个系统的结构如图9-11所示。

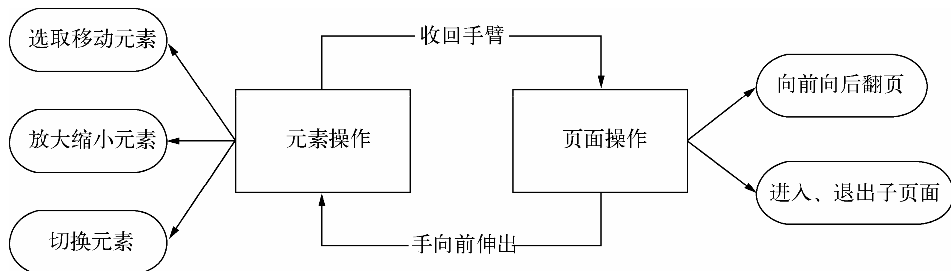



图9-11　整体交互设计示意图

## 9.4.3　Kinect自然交互小结



目前，Kinect的应用开发尚处于起步阶段，受设备和技术的限制仍然比较明显。在前几节中，我们介绍了一种较为成熟的解决方式，即利用Toolbox类库实现基础识别功能，然后对应不同需求设计具体的交互方式。在这种解决方案中，交互设计的部分尤为重要，应该遵循以下几项基本原则。

- 最小相互干扰：无论识别算法多么精确，都会存在一定的误识别现象，尤其是相似的动作姿态。所以在交互设计的时候应该尽量避免相似的动作，尤其是避免在同一时刻用相似的动作表示不同的操作。
- 最少识别时间：对于手势识别和语音识别的高误识别率来说，如果长时间开启，会将很多小动作、小声音误识别为目标操作。因此应该最大限度地缩小这种识别的持续时间，最好是利用姿态识别，仅仅在需要进行手势识别的时候才开启。

- 最大契合动作习惯：这一点正是自然交互的要求，让每一个交互动作都符合日常的动作习惯，比如抬腿表示走动、左右挥手表示切换等。

这样可以最大限度地提高Kinect应用的交互体验，并避免误操作、误识别。不过我们相信，随着Kinect设备性能的提高以及开发技术的发展，这些设计上的限制也会慢慢消失。

## 9.5　演示系统简介

前面几节着重介绍了虚拟演示系统是如何利用Kinect SDK完成各个功能模块的，但是其演示功能本身并不太依赖Kinect技术，仅需要按照一定逻辑进行绘图即可。因此该项目设计了一个基于XNA游戏引擎的演示框架来完成程序的基本功能。

整个演示框架分为三层进行管理：第一层是每个页面上的单独可视元素，类似于传统幻灯片每一页中的图片和文本框；第二层是由多个元素以及背景组成的页面，即“一张幻灯片”；第三层是由多个页面组成的演示内容整体。不同层有不同的管理逻辑，但是Kinect的控制流程是贯穿始终的，在每一层的操作中都会用到。

由于不同应用所使用的框架及开发方法不尽相同，可重用性很低，因此大家在阅读时不必将重点过多地放在代码上。另外，由于Kinect SDK以及现有的大多数识别框架都使用事件回调机制来处理相应的动作，这和轮询类框架相冲突，因此需要将其封装起来，便于轮询调用。下面我们将分析该应用在XNA框架下的Kinect封装类。

### 9.5.1　预备知识

由于虚拟演示系统的界面是采用XNA 4.0绘制的，整个系统的控制逻辑也包含在该框架内，所以为了便于读者理解代码，本节将简单介绍XNA框架。

#### 1. XNA框架简介

Microsoft XNA Framework是由微软发布的，用于辅助电脑游戏开发、电脑软件开发及管理的集成开发环境。XNA将游戏设计人员从“反复刻版编程”中解放出来，使得游戏编程更加容易和快捷。它能够自行实现检查显卡、创建设备、消息事件处理、纹理导入等工作，而程序员要做的只是编写游戏逻辑代码。它是基于.NET Framework之上的框架，以托管代码的方式来运行。另外还包含一些专用于游戏开发的类库，可以在指定的平台上使代码重用达到最大效果。

XNA框架包含了所有用作游戏编程的低阶技术，由此，游戏开发人员就可以专注于游戏内容开发而不用关心游戏在不同平台上的移植性问题。使用XNA平台开发的游戏，可以在所有支持XNA的设备上运行，比如PC、Xbox以及Windows Phone 7。此外，XNA框架还内置了一些工具，比如XACT，来辅助游戏内容、视觉和听觉效果的开发以及像真度很高的模型制作。

#### 2. XNA轮询框架

XNA框架的一个最大特点就是采用轮询方式来维护游戏的逻辑及绘图。与其他GUI框架的事件回调机制相比，轮询方式的效率更高。在XNA框架中，无论是`Game`类还是`Component`类，都遵循着一个固定的运行顺序，如图9-12所示。

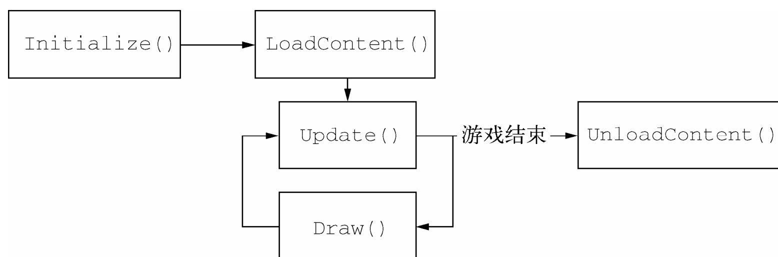



图9-12 XNA框架示意图

可以看到，每一个组件都要先在`Initialize()`函数中完成初始化过程，并通过`LoadContent()`函数加载所需资源，然后游戏以及组件就会反复执行`Update()`和`Draw()`方法来完成逻辑和绘图工作，直到游戏结束，执行`UnloadContent()`函数释放资源。虚拟环境演示系统也是建立在这种轮询的框架内的。

### 9.5.2 Kinect状态类

Kinect组件和其他组件最重要的一个对象就是Kinect状态类。程序将使用该类来传递Kinect的识别结果及数据。非Kinect组件开发者无需了解其调用的具体细节，只要使用该类的数据即可完成对应的功能，而Kinect开发者只需尽可能高效地完成该类所需的数据即可。因此，可以说Kinect状态类是连接整个项目最为重要的纽带。

#### 1. 创建Kinect状态类

首先在Visual Studio 2010中新建一个C#类，命名为“NuiState.cs”。我们将在这个类中完成整个Kinect状态的封装，如图9-13所示。

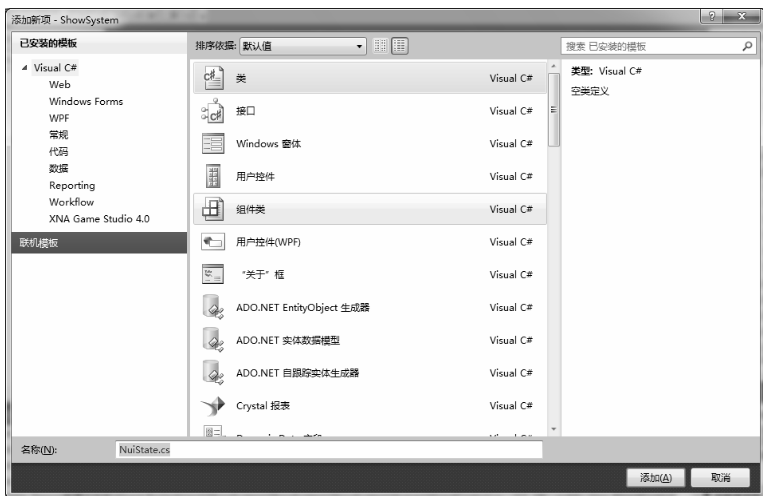



图9-13 在Visual Studio 2010中新建Kinect状态类

### 2. 加入所有需要传递的数据

我们需要在该类中添加之前提到的动作识别结果和拖曳所需的左右手位置等，相关代码如下：

```
public class NuiState
{
    public enum GestureStates
    {
        None,
        SwipeToLeft,
        SwipeToRight
    }

    public enum PostureStates
    {
        None,
        LeftHello,
        RightHello,
        LeftOverHead,
        RightOverHead,
        LeftSideWard,
        RightSideWard,
        BothReachOut,
        LeftReachOut,
        RightReachOut,
        HandsJoined,
        RightAtShoulder,
        LeftAtShoulder,
    }

    private bool iscatching;
    public bool IsCatching{ get;set; }
    {
        get { return iscatching; }
        set { iscatching = value; }
    }

    private bool isgesturing;
    public bool IsGesturing
    {
        get { return isgesturing; }
        set { isgesturing = value; }
    }

    private GestureStates gestureState;
    public GestureStates GestureState
    {
        get { return gestureState; }
        set { gestureState = value; }
    }

    private PostureStates postureState;
    public PostureStates PostureState
```

```
        {
            get { return postureState; }
            set { postureState = value; }
        }

        private Vector2 leftPosition;
        public Vector2 LeftPosition
        {
            get { return leftPosition; }
            set { leftPosition = value; }
        }

        private Vector2 rightPosition;
        public Vector2 RightPosition
        {
            get { return rightPosition; }
            set { rightPosition = value; }
        }
        public NuiState(NuiState e)
        {
            this.iscatching = e.IsCatching;
            this.isgesturing = e.IsGesturing;
            this.isListening = e.isListening;
            this.leftPosition = e.leftPosition;
            this.rightPosition = e.rightPosition;
            this.gestureState = e.GestureState;
            this.postureState = e.PostureState;
        }
        public NuiState()
        {
        }
    }
```

上述代码首先声明了动态动作以及静态姿态两种识别结果的枚举变量，这样不仅直观地显示所要处理的结果，同时也保证了高效性。

之后声明的`iscatching`和`isgesturing`布尔值用来区分上一节提到的两种交互阶段，`gestureState`和`postureState`记录当前帧的动作和姿态的识别结果（姿态识别阶段和元素操控阶段），最后的`leftPosition`和`rightPosition`表示左右手相对于屏幕的位置。

至此，全部的数据都已经封装到了`NuiState`类中。整个项目的组件都可以通过它来传递Kinect得到的数据。



### 9.5.3 Kinect轮询类

单凭状态类并不能完全满足轮询框架的需求。在一些情况下，轮询框架不仅要知道当前帧的状态，还要了解上一帧的状态，如图9-14所示。

事件回调机制是在需要处理的事件发生时抛出事件直接进行处理的，而在轮询框架下，只有知道了上一帧以及当前帧的状态才能判断出是否发生了状态改变。因此，为了便于对不同的组件进行调用，还需要完成一个轮询类来简化和重用Kinect状态轮询的过程。

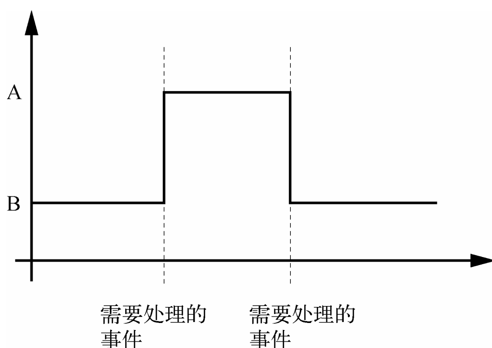



图9-14　状态改变示意图

**1. 创建Kinect轮询类**

同之前的步骤一样，新建一个C#类，命名为“KinectControl.cs”，Kinect状态轮询的过程将全部在这个类中完成。

**2. 实现Kinect轮询类**

系统需要在轮询类中完成两个任务，建立接口便于轮询框架控制每一次轮询的开始和结束，另外还要判断是否发生了某种状态改变事件。轮询类的相关代码如下：

```
public class KinectControl
{
    NuiState preState;
    public NuiState PreviousState
    {
        get { return preState; }
    }
    NuiState State;
    public NuiState CurrentState
    {
        get { return State; }
    }
    Nui nui;

    public KinectControl(Nui nui)
    {
        this.preState = null;
        this.nui = nui;
    }

    public void Begin()
    {
        preState = new NuiState(State);
        State = nui.CurrentState;
    }

    public bool TurnGestureState(NuiState.GestureStates e)
    {
        if (preState.GestureState != e && State.GestureState == e)
            return true;
```

```
    else
        return false;
}

public bool IsGestureState(NuiState.GestureStates e)
{
    if (State.GestureState == e)
        return true;
    else
        return false;
}

public bool StopGestureState(NuiState.GestureStates e)
{
    if (preState.GestureState == e && State.GestureState != e)
        return true;
    else
        return false;
}

public bool TurnPostureState(NuiState.PostureStates e)
{
    if (preState.PostureState != e && State.PostureState == e)
        return true;
    else
        return false;
}

public bool IsPostureState(NuiState.PostureStates e)
{
    if (State.PostureState == e)
        return true;
    else
        return false;
}

public bool StopPostureState(NuiState.PostureStates e)
{
    if (preState.PostureState == e && State.PostureState != e)
        return true;
    else
        return false;
}

public bool IsHandsSpliting()
{
    float preLen=(preState.LeftPosition-preState.RightPosition).Length();
    float curLen=(State.LeftPosition-State.RightPosition).Length();

    if (curLen - preLen > 0.0f)
        return true;
    else
        return false;
}

public bool IsHandsMerging()
```

```
        {
            float preLen = (preState.LeftPosition - preState.RightPosition).Length();
            float curLen = (State.LeftPosition - State.RightPosition).Length();

            if (preLen - curLen > 0.0f)
                return true;
            else
                return false;
        }
    }
```

在这个类中，我们首先声明了两个NuiState对象：preState和State，用来保存当前帧和上一帧的状态。 然后使用Begin()方法来更新当前帧的数据，先将上一帧的数据复制到preState中，再获取当前帧的数据。

后面的几个函数都是用来处理状态改变事件的，它们的具体用途如下所示。

- TurnGestureState：检测动作是否在当前帧改变到了某一状态。
- IsGestureState：检测当前帧动作是否为某一状态。
- StopGestureState：检测动作是否在当前帧终止。
- TurnPostureState：检测姿态是否在当前帧改变到了某一状态。
- IsPostureState：检测当前帧姿态是否为某一状态。
- StopPostureState：检测姿态是否在当前帧终止。
- IsHandsSpliting/IsHandsMerging：检测双手位置是否分开或合并。

以上函数的功能均为比较上一帧和当前帧的数据，并返回相应的布尔值。至此，我们就完成了一整套可以复用的框架，便于其他轮询框架的组件调用Kinect获得的数据结果。在这一框架中，其他组件仅需在轮询方法中分别调用KinectControl类中的begin()函数和相应的状态判断函数，即可得到所需的结果。

### 9.5.4　演示框架小结

在轮询框架中使用Kinect数据是开发者经常遇到的情况，另外，在软件开发的过程中，不可能所有的开发者都了解Kinect SDK的使用方法，因此设计一个合理且高效的接口也是必须要完成的工作。该项目提出了一种设计模式，即封装Kinect的全部操作，仅用一个状态类来传递所需的数据，同时对轮询框架编写可复用的代码来简化开发过程，如图9-15所示。

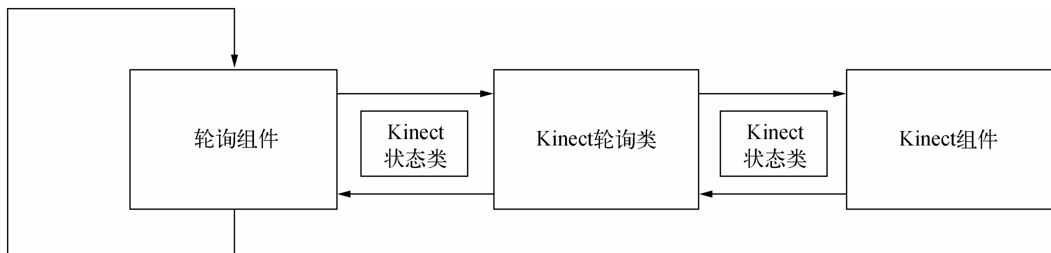



图9-15　Kinect组件与轮询组件交互方式

## 9.6 小结

在本章中，我们一起分析了基于Kinect SDK的应用，学习了它是如何利用SDK来实现所需功能的。其中，抠取人物彩色图像的实现方法较为新颖，灵活地运用了SDK提供的数据，而手势姿态以及语音识别的过程属于常规的需求，我们使用了一个第三方的开源类库来完成，节省了开发成本。但这并不代表这个项目的技术实现就是最优秀的，它仅仅是一个示例、一个引子，目的是为了启发大家利用Kinect SDK开发出更多有趣且实用的应用。

第 10 章

# Kinect虚拟放风筝项目的实现

风筝文化是我国重要的非物质文化遗产。2006年5月20日，风筝制作技艺被列入我国第一批国家级非物质文化遗产名录。但由于生活节奏加快，活动场地有限，放风筝的人越来越少。另外，由于老手工艺人不断减少，许多风筝制作技艺逐渐失传，风筝文化有着濒临绝迹的危机，急需得到人们的重视和保护。

那么如何使传统的风筝文化以一种新颖的方式得到更好的传承呢？本章介绍的Kinect虚拟放风筝项目就提出了一种方案，将微软最新的Kinect姿势识别技术与风筝文化结合，提供一种新的虚拟放风筝体验，实现风筝的室内放飞。这有益于风筝文化的传播以及非物质文化遗产的数字化保护。

## 10.1 Kinect 虚拟放风筝项目简介

中国的风筝文化源远流长，但目前放风筝活动都在室外进行，极易受场地、天气、季节等因素的影响，这使得放飞活动受到限制，也阻碍了风筝文化的发扬和传播。本项目结合了微软最新推出的体感人机交互设备Kinect，使大家可以在室内“放风筝”，并且还在项目中穿插了各种有关风筝文化的介绍。虚拟放风筝项目示意图如图10-1所示。

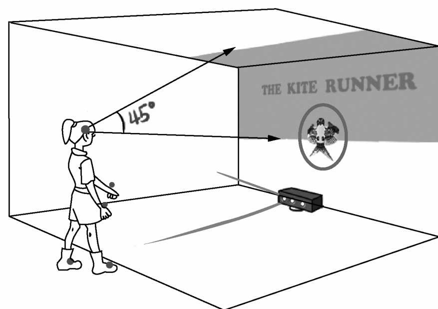



图10-1 虚拟放风筝项目示意图

Kinect虚拟放风筝项目是由清华大学美术学院的学生提出，并和微软亚洲研究院合作完成的。本项目恰到好处地利用了由Kinect for Windows SDK提供的骨骼追踪数据，使玩家可以通过真实放飞的肢体动作实现对虚拟风筝的控制。项目中的风筝图像、场景图像以及所有UI元素都是由清华大学美术学院的学生手绘设计完成的，其中有关风筝文化的介绍也全部由他们调研整理。这使得玩家在体验虚拟风筝的同时，还能了解风筝文化，获得艺术熏陶。

## 10.2　技术实现概述

总的来说，Kinect虚拟放风筝项目在技术实现方面分为两部分：玩家姿势的识别和风筝动画的设计。玩家姿势的识别主要利用了Kinect for Windows SDK提供的骨骼追踪数据，通过特定骨骼点的空间坐标信息来设计和判定控制姿势。整个软件的交互操作控制以及放飞模块对风筝的控制，都是由玩家通过姿势和手势来完成的。而由玩家姿势触发的风筝动画则是通过WPF动画来实现的，其中风筝模型以及放飞场景都是由WPF 3D绘制而成。

由于本项目的目的是传播风筝文化，更多的是展示文化内容，并实现用户与软件间的交互，因此我们选用了WPF框架。这样可以很方便地调用WPF的已有控件，同时完成3D绘图和一些简单的动画效果。需要注意的是，要尽量简化3D模型，以提高程序的性能。

由于前面的大部分实例程序都用到了WPF的相关知识，因此本章在介绍技术细节实现时，对于涉及WPF相关知识的部分只做简略介绍。如果想要深入了解WPF框架，请读者参阅讲解WPF应用开发的相关图书。

本项目的亮点在于结合了微软Kinect的体感交互技术，利用Kinect提供的骨骼数据，通过设计和识别控制姿势，使玩家能够更加自然地与应用程序进行交互。后面几节会详细介绍本项目涉及的每个姿势的实现细节。另外，由于我们已经在前几章中详细讲解了骨骼数据的一些基本知识，因此本章将重点介绍如何使用骨骼数据进行姿势的识别和判定。其中的方法不一定是最好的解决方案，这里仅提供一些思路，帮助读者从应用的角度进一步理解体感交互，同时也希望读者在此基础上提出自己的想法和解决方案。

## 10.3　玩家姿势的设计和识别

实际上，本项目就是将玩家特定的姿势映射到虚拟场景中对应的动画，因此玩家姿势的设计和准确识别至关重要。



### 10.3.1　玩家姿势的设计

在本项目中，我们主要设计了4个关键动作来模拟放飞风筝的真实体验，并通过对这些姿势动作的匹配识别来驱动风筝模型做出相应的动画。以下姿势动作主要是由清华大学美术学院的学生设计和绘制。

- 双手高举过头，风筝开始起飞，如图10-2所示，这是在进入放飞模块前要做的第一个动作。之所以采用这种姿势启动程序，主要是考虑到在多数情况下人的双手都低于头部，这样其他多余的姿势动作不会对启动命令造成干扰。

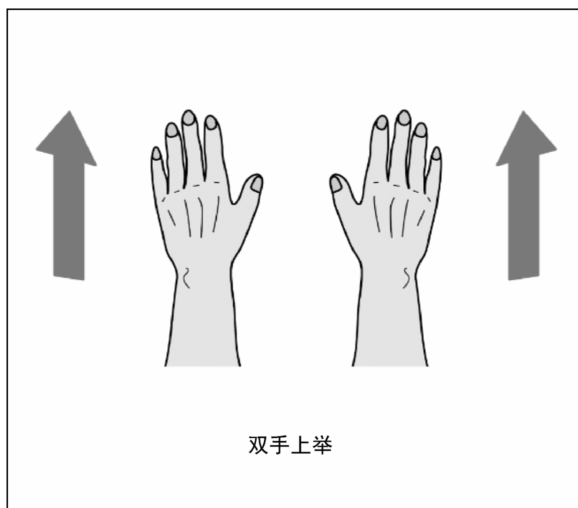



图10-2　举手姿势示意图

- 右手举过肩，并做振臂动作，风筝会飞高、飞远，如图10-3所示。这个动作结合了实际放飞时人们最常用的动作——举手来回拉风筝线。




图10-3　振臂动作示意图

- 双脚原地踏步，视野中的场景会随之而动，这也是为了模拟真实的情况，如图10-4所示。脚步的移动更能增加放风筝的真实感。

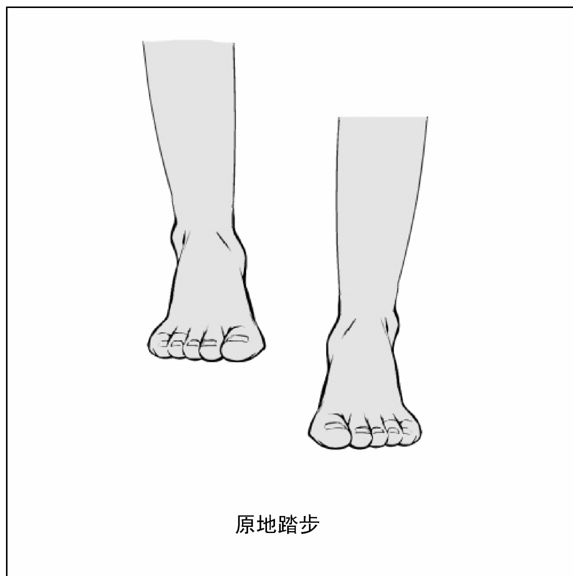



图10-4 踏步动作示意图

- 双手在身前交叉转动，会触发收线效果，即风筝飞低、飞近，如图10-5所示。这个动作结合了放风筝时收线的动作——双手交替转动线轴收线。

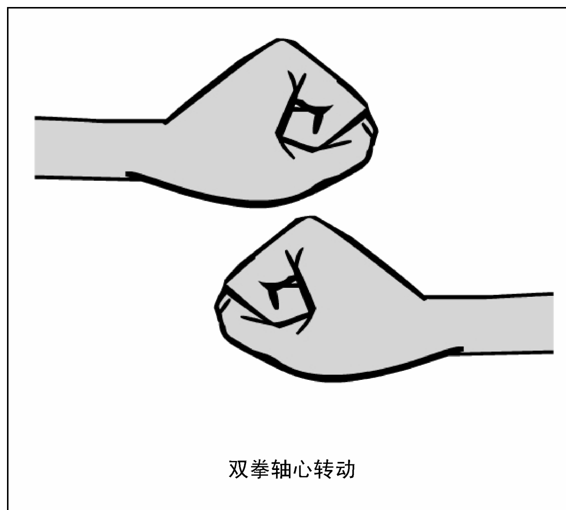



图10-5 转手动作示意图

这些姿势和动作都是根据真实的放飞体验而设计的。同样，读者也可以根据自己的项目需求进行设计，当然也要同时考虑实现难度与识别稳定性。

### 10.3.2　玩家姿势识别的实现

玩家姿势的识别主要是利用Kinect for Windows SDK获取到的骨骼追踪数据来实现的。骨骼追踪数据的数据结构以及获取方式在前面已详细介绍过，这里就不再赘述了，只介绍如何通过获取的骨骼数据来识别这4种姿势和动作。

#### 1. 双手高举过头姿势的识别

识别此姿势需要用到头部、左手和右手3个节点的信息，首先使用LINQ语句从每一帧的骨骼数据中提取出这3点的信息。接下来判断左、右手节点的位置信息Y坐标是否都高于头部节点的Y坐标，如果成立则返回真值。同时也可以加入一个确定的阈值来进行该姿势识别的微调。相关代码如下：

```
private bool CheckBothHandsOverHead(Skeleton s)
{
    Joint lefthandJoint = (from j in s.Joints
                           where j.JointType == JointType.HandLeft
                           select j).FirstOrDefault();
    Joint righthandJoint = (from j in s.Joints
                            where j.JointType == JointType.HandRight
                            select j).FirstOrDefault();
    Joint headJoint = (from j in s.Joints
                       where j.JointType == JointType.Head
                       select j).FirstOrDefault();

    if (lefthandJoint.Position.Y > headJoint.Position.Y &&
        righthandJoint.Position.Y > headJoint.Position.Y)
    {
        return true;
    }
    return false;
}
```

#### 2. 右手举过肩，并做振臂动作的识别

识别此动作只需用到右手和右肩节点的信息。首先判断右手节点的Y坐标是否高于右肩节点的Y坐标，在此条件成立的情况下判断振臂动作。判断振臂动作时，需要记录两次右手节点Z坐标和右肩节点Z坐标的差值，阈值`MIN_ShoulderRight_HandRight_Z`的大小是通过测试得到的，可以进行适当的调整。相关代码如下：

```
private bool CheckShakeRightHand(Skeleton s)
{
    Joint righthandJoint = (from j in s.Joints
                            where j.JointType == JointType.HandRight
                            select j).FirstOrDefault();
    Joint shoulderRightJoint = (from j in s.Joints
                                where j.JointType == JointType.ShoulderRight
```

```
                            select j).FirstOrDefault();
    if (!(righthandJoint.TrackingState == JointTrackingState.Tracked &&
        shoulderRightJoint.TrackingState == JointTrackingState.Tracked))
        return false;
    //右手高于肩
    if (righthandJoint.Position.Y > shoulderRightJoint.Position.Y)
    {
        if (Math.Abs(shoulderRightJoint.Position.Z - righthandJoint.Position.Z) <
            MIN_ShoulderRight_HandRight_Z)
            shakeFlag = true;
        if (Math.Abs(shoulderRightJoint.Position.Z - righthandJoint.Position.Z) >
            MIN_ShoulderRight_HandRight_Z && shakeFlag)
        {
            shakeFlag = false;
            return true;
        }
    }
    return false;
}
```

### 3. 双脚原地踏步动作的识别

识别该动作需要用到左膝和右膝两个节点的信息，玩家在原地踏步时这两点的Z坐标相对位置的变换比较显著。同样地，识别此动作时，也需要记录两次的数据以进行判断，阈值MIN_KneeLeft_KneeRight的大小也可以通过测试得到。另外，还能通过其他节点在踏步过程中值的变化来进行识别，比如左、右脚节点。

```
private bool CheckRunPosture(Skeleton s)
{
    Joint rightkneeJoint = (from j in s.Joints
                            where j.JointType == JointType.KneeRight
                            select j).FirstOrDefault();
    Joint leftkneeJoint = (from j in s.Joints
                           where j.JointType == JointType.KneeLeft
                           select j).FirstOrDefault();
    if (!(rightkneeJoint.TrackingState == JointTrackingState.Tracked &&
        leftkneeJoint.TrackingState == JointTrackingState.Tracked))
        return false;
    if (leftkneeJoint.Position.Z - rightkneeJoint.Position.Z >
        MIN_KneeLeft_KneeRight_Z)
    {
        runFlag = true;
    }
    if ((rightkneeJoint.Position.Z - leftkneeJoint.Position.Z >
        MIN_KneeLeft_KneeRight_Z) && runFlag)
    {
        runFlag = false;
        return true;
    }
    return false;
}
```

**4. 双手在身前交叉转动动作的识别**

识别此动作需要用到左、右手节点的信息，通过测试观察发现，在玩家做此动作时，双手的X坐标差值保持在一定范围内，而双手的Y坐标差值则有显著的周期性变化。在进行此动作的识别时，可以利用这些特征首先判断左、右手节点的X坐标的差值是否处于一定的阈值范围内，此条件成立才进行下一步判断，这一条件能有效地提高此动作识别的准确性和稳定性。进一步判断双手Y值的变化时，同样也需要通过两次判断得出结论。相关代码如下：

```
private bool CheckHandsCross(Skeleton s)
{
    Joint lefthandJoint = (from j in s.Joints
                           where j.JointType == JointType.HandLeft
                           select j).FirstOrDefault();
    Joint righthandJoint = (from j in s.Joints
                            where j.JointType == JointType.HandRight
                            select j).FirstOrDefault();
    if (!(lefthandJoint.TrackingState == JointTrackingState.Tracked &&
        righthandJoint.TrackingState == JointTrackingState.Tracked))
        return false;
    //双手靠近
    if (Math.Abs(lefthandJoint.Position.X - righthandJoint.Position.X) <
        MAX_HandLeft_HandRight_X)
    {
        if (righthandJoint.Position.Y - lefthandJoint.Position.Y >
            MIN_HandLeft_HandRight_Y)
            crossFlag = true;
        if (lefthandJoint.Position.Y - righthandJoint.Position.Y >
            MIN_HandLeft_HandRight_Y && crossFlag)
        {
            crossFlag = false;
            return true;
        }
    }

    return false;
}
```

同一姿势的识别方法不止一种，读者可以根据自己的项目需求进行评估，使用准确性和稳定性最好的方法。其中的阈值则是经过多次试验得到的经验数据，需要读者根据具体情况进行选择。

## 10.4　自然交互按钮和光标的实现

传统意义上，应用程序和用户之间最直接的交互少不了按钮和光标，而Kinect应用要求用户使用体感控制，其交互方式与传统方式有很大的不同。这种“自然交互”式的按钮和光标也是Kinect应用开发中使用频率相当高的组件，因此本节将着重介绍这种交互方式的简单实现方法：将光标元素与手节点绑定，通过移动手的位置来移动光标；当光标在想要选择的按钮上悬停几秒后，则触发按钮单击的事件。

### 10.4.1 自定义光标

在项目中添加一个用户控件，即“User Control(WPF)”，命名为“CursorUserControl”，如图10-6所示。Visual Studio 2010会自动为项目添加“CursorUserControl.xaml”和“CursorUserControl.xaml.cs”两个文件。

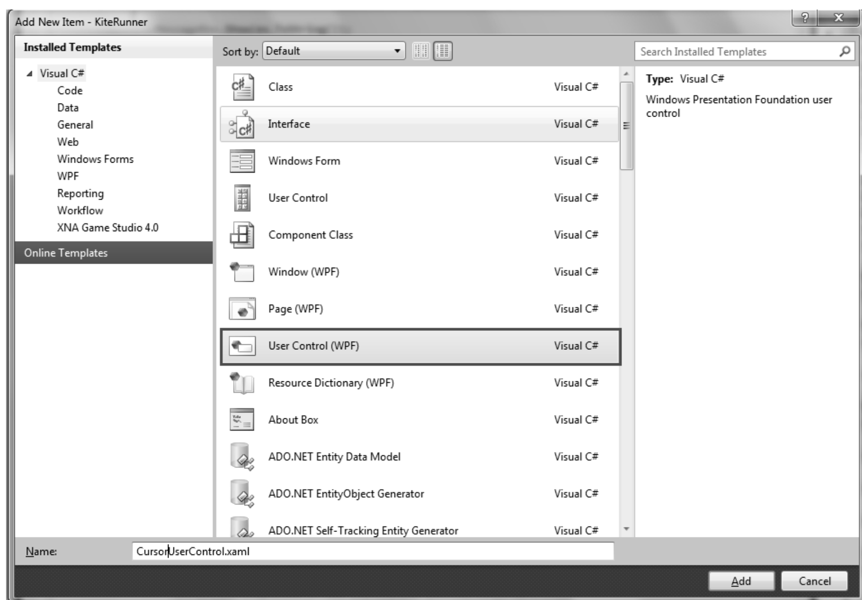



图10-6 新建用户控件

在CursorUserControl.xaml文件中定制光标的样式。针对本项目，当光标悬停在按钮上时，会触发蓝色阴影注满光标的动画，设计代码如下所示。

```
<Grid x:Name="animateGrid" RenderTransformOrigin="0.5,0.5">
    <Image HorizontalAlignment="Left"
           Margin="0"
           Name="image1"
           Stretch="Uniform"
           Source="Images/Button/kiteCursor.png" VerticalAlignment="Top" />
    <Ellipse HorizontalAlignment="Left"
             Margin="0"
             Name="ellipse1"
             Stroke="Blue"
             VerticalAlignment="Top"
             Width="50"
             Height="50"/>
    <Ellipse HorizontalAlignment="Left"
             Margin="0"
             Name="animateEllipe"
             Stroke="Blue"
```

10

```
            VerticalAlignment="Bottom"
            Width="50"
            Height="0"
            Fill="Blue"
            Opacity="0.5"/>
</Grid>
```

Image控件设置光标的配图，第一个Ellipse控件用来圈出光标的轮廓，第二个Ellipse控件覆盖在第一个Ellipse上，填充为半透明的蓝色，高度初始化为0，逐渐注满阴影的动画其实就是逐渐增加第二个Ellipse的高度值。正常情况和注满阴影的光标设计效果如图10-7所示。

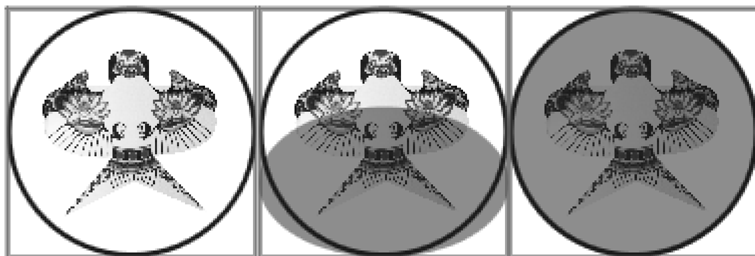


图10-7　光标设计图

自定义光标控件的后台代码值，将右手节点的位置坐标关联到光标的位置属性上即可。相关代码如下：

```
public void SetPosition(Point pos)
{
    Canvas.SetTop(this, pos.Y - 25);
    Canvas.SetLeft(this, pos.X - 25);
}
```

## 10.4.2　自定义按钮

同前面一样，在项目中添加一个用户控件，命名为“HoverButton”，代表自定义按钮控件。Visual Studio 2010会自动为项目添加“HoverButton.xaml”和“HoverButton.xaml.cs”两个文件。

在HoverButton.xaml中定制按钮的样式。针对本项目，按钮将全部以图片表示，因此只需添加一个Image控件用于容纳图片，填充的图片需要动态绑定。相关代码如下：

```
<Grid>
    <Image Source="{Binding ImageSource}" Stretch="Fill"/>
</Grid>
```

接着在后台代码（即HoverButton.xaml.cs文件）中添加单击事件处理、动画等功能。从文件自动生成的代码中可以看出，HoverButton继承于UserControl类。

以下是按钮动画的编写步骤及单击事件的绑定和响应过程。

(1) 为添加的Image控件提供对外的接口，以便于在程序中添加HoverButton时设置相应的图片源。代码如下所示，注意属性名称要与控件绑定的名称一致。

```
public string ImageSource
{
    get { return (string)this.GetValue(ImageSourceProperty); }
    set { this.SetValue(ImageSourceProperty, value); }
}
public static readonly DependencyProperty ImageSourceProperty =
    DependencyProperty.Register("ImageSource", typeof(string),
                                typeof(HoverButton),
                                new PropertyMetadata(""));
```

(2) 定义单机事件的委托，相关代码如下：

```
public delegate void ClickHandler(object sender, EventArgs e);
public event ClickHandler Click;
```

(3) 判断光标是否悬停在按钮范围内，先取到光标中心点的坐标，然后通过按钮的位置和宽高计算按钮的表示范围，最后判断光标中心点是否在按钮范围内。相关代码如下：

```
public bool CursorInButton(FrameworkElement MyCursor)
{
    Point CursorPos = MyCursor.PointToScreen(new Point());

    double CursorMidX = CursorPos.X + (MyCursor.ActualWidth / 2);
    double CursorMidY = CursorPos.Y + (MyCursor.ActualHeight/ 2);

    Point HoverButPos = this.PointToScreen(new Point());
    double HoverButLeft = HoverButPos.X;
    double HoverButRight = HoverButLeft + this.ActualWidth;
    double HoverButTop = HoverButPos.Y;
    double HoverButBottom = HoverButPos.Y + this.ActualHeight;
    if (CursorMidX < HoverButLeft || CursorMidX > HoverButRight ||
        CursorMidY < HoverButTop || CursorMidY > HoverButBottom)
        return false;
    else
        return true;
}
```

(4) 程序会不停地判断光标是否在按钮范围内。如果在，则触发一段动画，即蓝色阴影逐渐注满光标元素，时长为2秒。代码中注册了代表此动画完成的事件`CursorFill_Completed`，当光标悬停在按钮上的时间达到2秒，即动画完成时会触发该事件；如果光标在阴影注满之前就离开了按钮，则会结束动画，恢复光标状态，相关代码如下所示。(关于WPF动画实现的原理，这里就不展开介绍了，请读者参阅相关的WPF书籍。)



```
private Duration HoverDuration = new Duration(new TimeSpan(0,0,2));
private Duration ReverseDuration=new Duration(new TimeSpan(0,0,1));
public bool IsHovered(FrameworkElement MyCursor)
{
    if(CursorInButton(MyCursor))
    {
        BeginHover((CursorUserControl)MyCursor);
        return true;
    }
    else
```

```
        {
            EndHover((CursorUserControl)MyCursor);
            return false;
        }
    }

    private void BeginHover(CursorUserControl m_cursor)
    {
        try
        {
            if (IsHovering == false)
            {
                IsHovering = true;
                ScaleTransform scale = new ScaleTransform(1.1, 1.2);
                //MyGrid.RenderTransform = scale;
                m_cursor.animateGrid.RenderTransform = scale;
                CursorFill = new DoubleAnimation(m_cursor.animateEllipe.ActualHeight,
                    m_cursor.ActualHeight, HoverDuration);
                CursorFill.Completed += new EventHandler(CursorFill_Completed);
                m_cursor.animateEllipe.BeginAnimation(Canvas.HeightProperty,
                    CursorFill);
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.ToString());
        }
    }

    private void EndHover(CursorUserControl m_cursor)
    {
        if (IsHovering == true)
        {
            IsHovering = false;
            ScaleTransform scale = new ScaleTransform(1.0, 1.0);
            m_cursor.animateGrid.RenderTransform = scale;
            CursorFill.Completed -= CursorFill_Completed;
            CursorFill = new DoubleAnimation(m_cursor.animateEllipe.ActualHeight, 0,
                                             ReverseDuration);
            m_cursor.animateEllipe.BeginAnimation(Ellipse.HeightProperty, CursorFill);
        }
    }
```

(5) 前面提到，当光标悬停在按钮上的时间达到2秒，即阴影注满光标的动画完成后会触发动画完成事件，而动画完成事件则会触发单击按钮事件。相关代码如下：

```
    private void CursorFill_Completed(object sender, EventArgs e)
    {
        IsHovering = false;
        if (Click != null) Click(this, e);
        CursorFill.BeginAnimation(Ellipse.HeightProperty, null);
    }
```

至此，自定义的按钮和光标就完成了。读者也可以按照类似的方法，根据自己的需求和设计自定义光标和按钮。

## 10.5　风筝动画的实现

针对玩家做出的不同姿势和动作，风筝会以飞上、飞下等动画来响应，本项目中的风筝动画是通过WPF动画设计来实现的。简而言之，WPF动画本质上就是在一段时间间隔内修改依赖项属性值的一种方式，也就是说，WPF动画是基于时间和属性的动画。比如，为了增大和缩小一个按钮，可以在动画中修改按钮的宽度。

风筝放飞的场景设计为WPF 3D绘图，这里可以将风筝视为一个3D模型的对象，风筝的飞高、飞低其实就是改变风筝模型在3D场景中的Y坐标；同样，风筝的飞远、飞近就是改变风筝模型在场景中的Z坐标。此外，还会用到风筝模型的旋转轴和旋转角度属性。下面将通过几个动画来介绍具体的实现方法，其他动画大同小异。

### 1. 起飞动画的实现

风筝起飞动画其实就是风筝从地面瞬间飞高、飞远，如下列代码所示。其中，飞远动画修改的是风筝模型`KiteTranslateTransform3D`的`TranslateTransform3D.OffsetZProperty`属性值，该值越小，玩家视野中的风筝就越远，代码中的相应部分表示在0.5秒的时间内，该属性值由初始值0改变为–20；而飞高动画则是将风筝模型的`TranslateTransform3D.OffsetYProperty`属性值，在0.5秒内由初始值0改变为10。

```
private void StartAnimate()
{
    //飞远
    DoubleAnimation zAnimation = new DoubleAnimation();
    zAnimation.To = -20;
    zAnimation.Duration = TimeSpan.FromSeconds(0.5);
    KiteTranslateTransform3D.BeginAnimation(TranslateTransform3D.OffsetZProperty,
                                            zAnimation);

    //飞高
    DoubleAnimation yAnimation = new DoubleAnimation();
    yAnimation.To = 10;
    yAnimation.Duration = TimeSpan.FromSeconds(0.5);
    KiteTranslateTransform3D.BeginAnimation(TranslateTransform3D.OffsetYProperty,
                                            yAnimation);
}
```



当玩家振动右臂时，风筝飞高、飞远的动画跟起飞动画相似，只不过每次振臂，风筝模型相应属性值改变的幅度不一样。同样，玩家做收线动作时，风筝飞低、飞近的动画相当于把风筝飞高、飞远的动画反过来。

### 2. 晃动动画的实现

为了使风筝的动画更加生动，而不是生硬地飞高、飞低，当风筝起飞后，要让它不停地左右摆动。风筝晃动动画需要以(0,1,0)为旋转轴，并将风筝的旋转角度属性`AxisAngleRotation3D.`

AngleProperty从5改变到−5，时间间隔为1秒。将该动画的AutoReverse属性设置为true，可以使该动画完成之后自动反转，即属性值再从-5改变到5。实现代码如下所示：

```
private void SwingAnimate()
{
    swingAnimateFlag = false;
    //myAxis.Angle = 5.0;
    this.swingAnimation = new DoubleAnimation();
    swingAnimation.From = 5;
    swingAnimation.To = -5;
    swingAnimation.Duration = TimeSpan.FromSeconds(1);
    swingAnimation.AutoReverse = true;
    this.swingAnimation.Completed += swingAnimation_Completed;
    KiteAxis.Axis = new Vector3D(0, 1, 0);
    KiteAxis.BeginAnimation(AxisAngleRotation3D.AngleProperty, swingAnimation);
}
private void swingAnimation_Completed(object sender, EventArgs e)
{
    swingAnimateFlag = true;
    this.swingAnimation.BeginAnimation(AxisAngleRotation3D.AngleProperty, null);
}
```

**3. 风筝坠落动画**

风筝坠落一般分为两个动作，即风筝旋转为头部朝下，并同时下落。其实现原理与前面的动画类似，相关代码如下：

```
private void DropKiteAnimate()
{
    if (flyingFlag == true)
    {
        flyingFlag = false;
        //翻转动画
        DoubleAnimation rolloverAnimation = new DoubleAnimation();
        rolloverAnimation.By = 180;
        rolloverAnimation.Duration = TimeSpan.FromSeconds(5);
        KiteAxis.Axis = new Vector3D(0, 0, this.KiteTranslateTransform3D.OffsetZ);
        //调整旋转轴线
        KiteAxis.BeginAnimation(AxisAngleRotation3D.AngleProperty, rolloverAnimation);

        //下落动画
        dropHeight = this.KiteTranslateTransform3D.OffsetY;
        dropAnimation = new DoubleAnimation();
        dropAnimation.To = 0;
        dropAnimation.Duration = TimeSpan.FromSeconds(5);
        dropAnimation.Completed += new EventHandler(dropAnimation_Completed);
        KiteTranslateTransform3D.BeginAnimation(TranslateTransform3D.OffsetYProperty,
                                               dropAnimation);
    }
}
```

这里仅介绍了几个主要动画的实现过程，要想在整个风筝的放飞过程中实现流畅体验还要辅以更加复杂的处理流程，这里就不作介绍了。毕竟本节的重点是讲述与Kinect体感交互技术相关的实现过程。

## 10.6　项目操作流程

本项目中的操作完全由用户的姿势和动作控制，包括选择、设置、放飞等，且仅支持一个玩家进行操作。首先将Kinect设备连接到电脑，在电脑上运行应用程序，玩家需站在Kinect的最佳识别区域内进行操作，大概距离Kinect设备1.5～2米。具体的操作流程如下所示。

(1) 首先进入欢迎页面，如图10-8所示。




图10-8　初始界面

通过挥动右手，将风筝样式光标移动到中间的风筝LOGO上，悬停2秒待阴影注满光标即可进入下一个页面——选择风筝页面，如图10-9所示。




图10-9　单击“开始”按钮

(2) 选择风筝页面如图10-10所示。进入该页面后，系统会自动播放关于风筝历史、文化的音频。此外，玩家还可以在这里选择风筝的样式，共有5类。




图10-10　选择风筝界面

当选中一种风筝样式后，音频会自动切换为介绍该样式风筝的内容，如图10-11所示。




图10-11　选中风筝

选择“立刻放飞”将进入风筝放飞页面，此时放飞场景中的风筝即为刚刚选中的样式，如图10-12所示。




图10-12　单击“立刻放飞”按钮

(3) 进入放飞页面后，玩家就可以使用前面介绍的姿势来控制放飞场景中的风筝了。图10-13、图10-14为风筝放飞和坠落的截图。

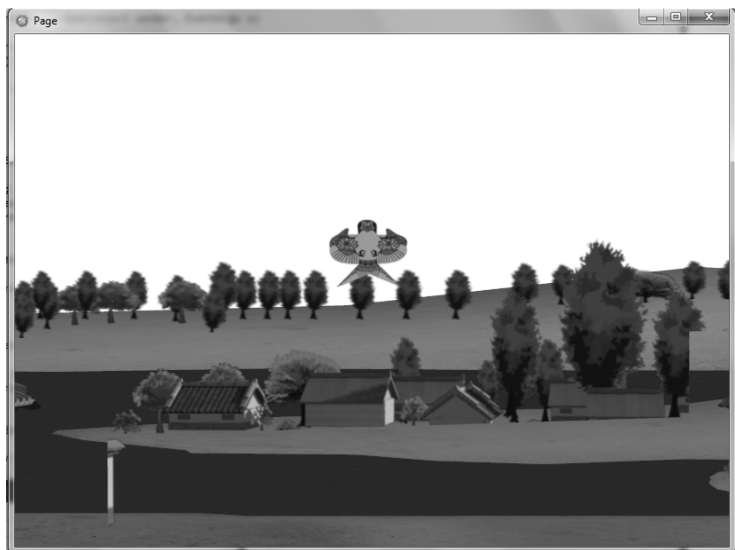



图10-13　放飞界面

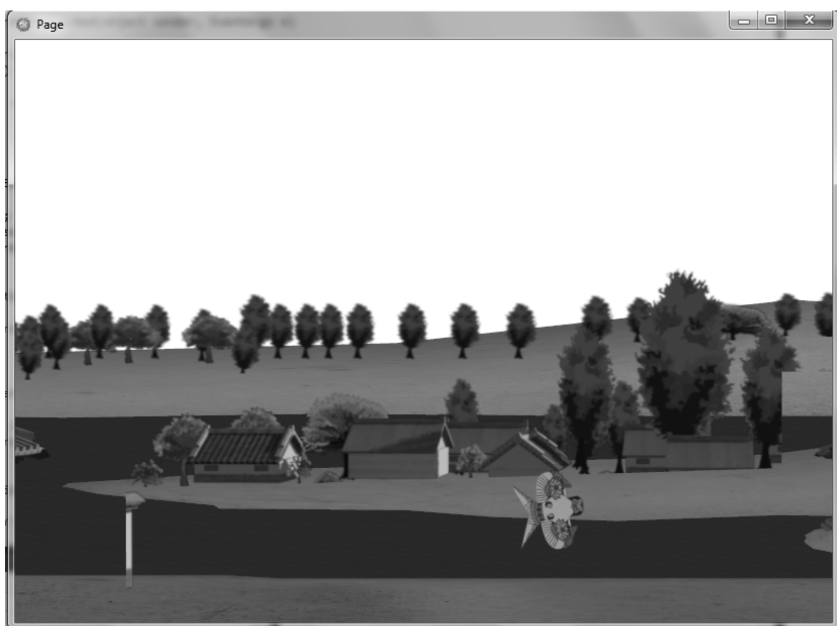


图10-14　风筝落下

(4) 当放飞结束后会跳出如图10-15所示的页面，玩家可以重新选择风筝或重玩一次。




图10-15　放飞结束界面

## 10.7 小结

Kinect虚拟放风筝项目旨在以一种新的形式对中国传统的风筝文化进行传播，其中微软最新的Kinect体感交互技术起到了关键作用。Kinect for Windows SDK支持人体骨架点的实时捕获，这使得玩家可以通过模拟实际放风筝时的姿势和动作来控制虚拟的风筝模型。新颖的技术和传统的文化相结合，必将吸引更多的年轻人加入到风筝文化的传播中来。

第 11 章

# Kinect全息显示

全息显示是利用已有的物体三维信息，利用光学干涉的原理，在现实场景中真实重现其三维图像的技术，可以使观察者身临其境。但是现有的解决方案成本高昂，同时还存在显示色彩少、分辨率低等不足之处。本章选取的项目另辟蹊径，巧妙地利用了Kinect SDK提供的骨骼点追踪功能，结合普通的显示屏或者投影仪即可实现全息显示的效果。

不仅如此，该项目还以保护文化遗产为主题，被应用在博物馆文物展示中，具有较高的社会意义。因此它也晋级了2012微软精英大挑战决赛，并获得了三等奖的优异成绩。

## 11.1 Kinect 全息显示简介

现今的博物馆中存在很多损毁、需要特别保护或者由于其他各种原因而无法实物展览的文物。针对这些文物，博物馆一般会采用虚拟方式向游客展览，即先对文物进行3D建模，将其全部的外观信息进行数字化记录，然后借助屏幕展示虚拟的文物。此外，还有很多旅游景点也在其网站上实现了类似于虚拟全景游览的功能，让大家足不出户就可以享受到旅行的快乐。但是这些方法存在一个共同的问题——真实感较差。虽然具有原始文物和景观的3D信息，但通过显示介质，比如电脑显示器、电视机屏幕、投影仪和广告牌的荧幕等显示出来的图像都局限于平面。无论我们从哪个角度观看，看到的都是相同的画面，就像一幅纸质的画，无法达到身临其境的效果。即便是现在热门的3D显示技术，也只是让左眼和右眼看到两个不同的平面而已，而这两个面实际上也是固定不变的。

因此，如何让人们在参观虚拟文物或网上博物馆时，如同走入虚拟世界，真切地感受到眼前物体的存在，就成为了亟待解决的问题。而该项目就是为了解决这一问题而尝试的一种体验方法。它采用了全息显示技术，根据文物已有的三维信息，全方位地进行显示，让观察者在任何位置都能看到相应图像，如同观察实物一样。通过这种方法，大幅增强了虚拟文物展览的游客体验，同时也消除了很多珍贵文物因种种原因不能以实体方式展现的缺憾。

## 11.2 技术实现概述

在讨论这个项目的技术实现之前，我们先来看这样一个场景。电影《碟中谍4》中，主角和

他的同伴潜入克林姆林宫。为了在不惊动警卫的情况下通过一个甬道，他们在甬道中支起了一整块幕布，并在上面投影出幕布后面的影像，同时根据甬道尽头警卫的眼睛位置实时更新投影图像，使他根本无法察觉幕布的存在，他所看到的影像和真实的甬道一样。这就是全息显示。

传统的全息显示实现方案，有的成本太高，有的分辨率低，有的则只能进行单色显示，这使得该技术很难在博物馆日常展览中得到应用。而该项目采用的技术方案在本质上和前面提到的电影中的技术相同，即跟踪使用者眼睛的空间位置，并在屏幕上绘制该视角下能看到的图像，达到以假乱真的效果。但与之不同的是，在跟踪使用者的眼睛位置时，使用了更加准确且实时性较高的Kinect技术。这样既实现了全息显示的效果，又避免了单色、分辨率低等显示效果差的问题，使整个方案廉价易行。Kinect全息显示技术实现示意图如图11-1所示。

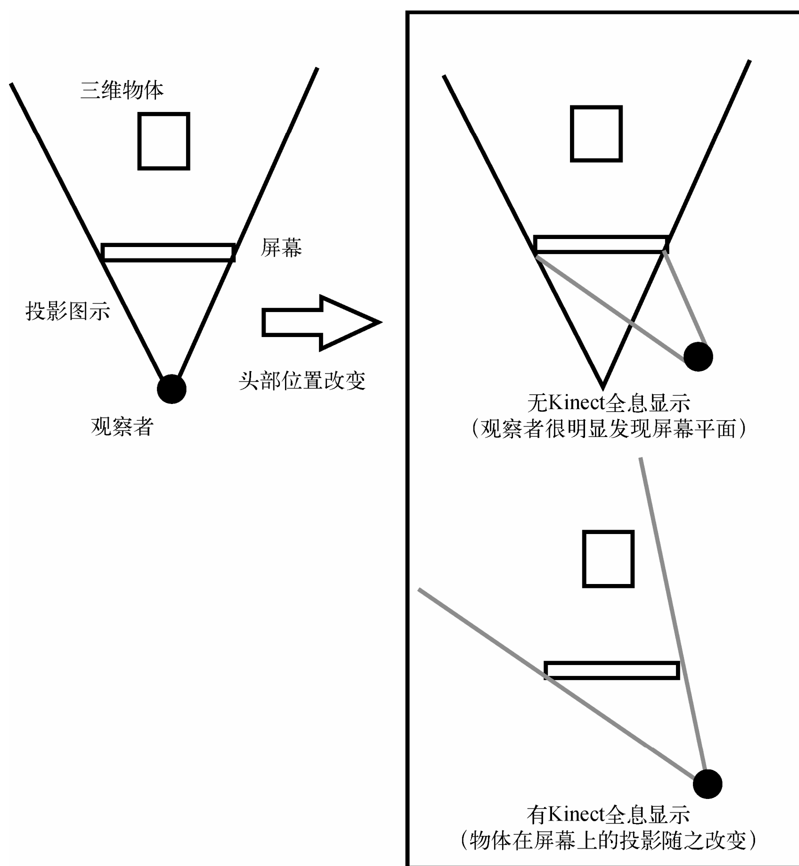



图11-1　Kinect全息显示技术实现示意图

为了达到这种效果，整个项目分三部分来实现。

(1) 利用Kinect设备捕捉观察者的头部位置，便于以后调整屏幕投射视角使用。

(2) 基本三维图形引擎绘制模型。

(3) 根据头部位置调整投影矩阵，绘制出最终的图像。

本章将在后面三节中分别对这三部分进行分析，并完成相应的代码实现。其中，三维图形引擎并不涉及Kinect相关知识，不是本书讲述的重点，因此这里仅介绍图形学的基础知识，以便读者理解整个项目的实现原理。另外，该项目代码是基于XNA 4.0框架编写的，不熟悉XNA框架的读者，可以先阅读本书的9.5节，或者通过其他相关书籍，了解XNA框架的基本运行方式，以便理解相关代码。

## 11.3　Kinect 捕捉头部坐标

从本节开始，我们正式进行项目技术实现的分析，首先要完成利用Kinect捕捉头部位置的组件。在Kinect SDK提供的骨骼数据支持下，可以很容易地完成这一功能。由于整个项目是基于XNA框架的，因此需要将此功能封装在一个XNA组件中。

### 11.3.1　创建用于捕捉头部位置的Kinect组件类

首先在Visual Studio 2010中新建一个C#类，然后选取Visual C#→XNA Game Studio 4.0→Game Component模板，将其命名为“KinectComponent.cs”，如图11-2所示。我们将在这个类中完成本节的全部工作。

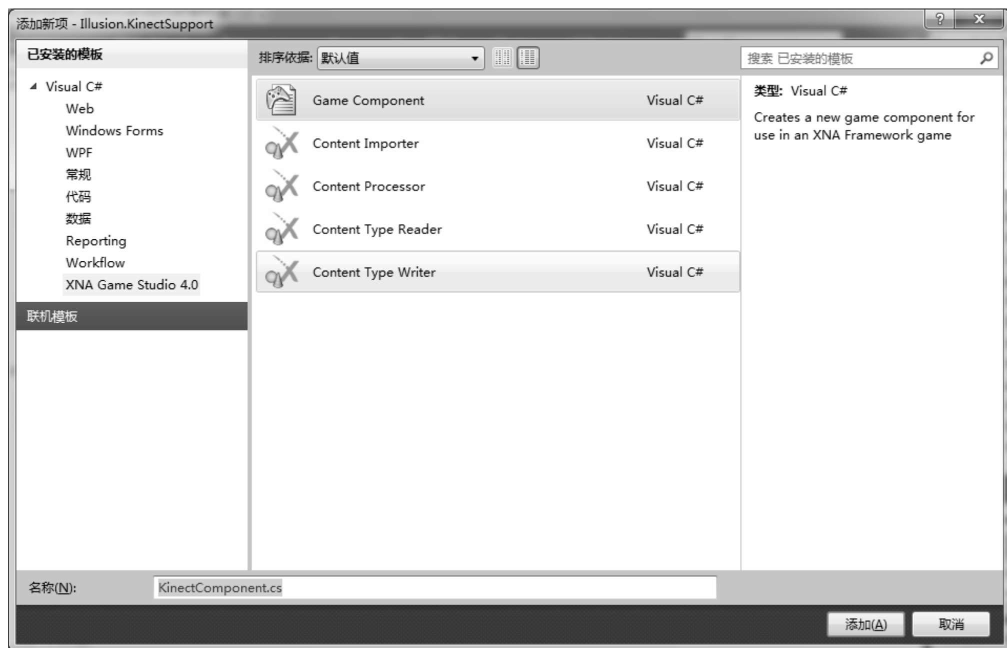



图11-2　在Visual Studio 2010中新建Kinect组件类

## 11.3.2 Kinect初始化以及头部位置获取

在这个组件类中，我们需要完成三部分的工作：Kinect初始化、获取各个合法的数据、获取头部位置。具体代码如下：

```
public class KinectComponent : Microsoft.Xna.Framework.GameComponent
{
    DateTime now;
    DateTime past;

    KinectSensor kinect;

    public int KinectID = -1;
    public Vector head = new Vector();

    public Vector3 Headposition
    {
        get
        {
            return new Vector3((float)head.X, (float)head.Y, (float)head.Z);
        }
    }

    public float HeadDepth
    {
        get
        {
            return (float)head.Z;
        }
    }

    public KinectComponent(Game game)
        : base(game)
    {
    }

    public override void Initialize()
    {
        if (Runtime.Kinects.Count > 0)
        {
            kinectSensor = (from sensor in KinectSensor.KinectSensors
                            where sensor.Status == KinectStatus.Connected
                            select sensor).FirstOrDefault();
            kinectSensor.SkeletonStream.Enable();
            kinectSensor.Start();

            kinectSensor.SkeletonFrameReady +=
                new EventHandler<SkeletonFrameReadyEventArgs>(kinect_SkeletonFrameReady);
        }
        base.Initialize();
    }
```

```
        void kinect_SkeletonFrameReady(object sender, SkeletonFrameReadyEventArgs e)
        {
            bool flag = false;
            SkeletonFrame skeletonFrame = e.OpenSkeletonFrame()
            foreach (var user in frame.Skeletons)
                if (user.TrackingID == KinectID &&
                    SkeletonTrackingState.Tracked == user.TrackingState)
                {
                    JointsCollection j = user.Joints;
                    head = j[JointID.Head].Position;
                    flag = true;
                    past = System.DateTime.Now;
                    break;
                }
            if (!flag)
            {
                now = System.DateTime.Now;
                if ((now - past).TotalMilliseconds > 1000)
                {
                    foreach (SkeletonData user in e.SkeletonFrame.Skeletons)
                        if (SkeletonTrackingState.Tracked == user.TrackingState)
                        {
                            JointsCollection j = user.Joints;
                            head = j[JointID.Head].Position;
                            KinectID = user.TrackingID;
                            break;
                        }
                }
            }
        }
    }
```

下面分别介绍这三项工作。

#### 1. Kinect初始化

因为只需要捕获头部位置，所以只要按照惯例开启骨骼跟踪模式，并注册骨骼回调函数kinect_SkeletonFrameReady即可。

```
    kinectSensor.SkeletonStream.Enable();
    kinectSensor.Start();
    kinectSensor.SkeletonFrameReady +=
        new EventHandler<SkeletonFrameReadyEventArgs>(kinect_SkeletonFrameReady);
```

#### 2. 获取合法骨骼数据并记录头部位置

由于投影屏幕的图像是由使用者的头部位置来决定的，因此需要保证对单一人物的持续性跟踪。加入标记变量KinectID，并在每一帧都进行检测，只筛选出需要跟踪的骨骼：

```
    if (user.TrackingID == KinectID &&
        SkeletonTrackingState.Tracked == user.TrackingState)
```

同时，在出现第一帧以及跟丢目标人物的情况下，需要重新记录TrackingID。为了避免个别骨骼帧中人物追踪失败而导致的重新记录，这里加入两个时间戳now和past，分别记录上一次识别到目标人物的时间和当前丢失帧的时间。如果相隔超过一秒就可以重新记录TrackingID了：

```
if ((now - past).TotalMilliseconds > 1000)
{
    foreach (SkeletonData user in e.SkeletonFrame.Skeletons)
        if (SkeletonTrackingState.Tracked == user.TrackingState)
        {
            JointsCollection j = user.Joints;
            head = j[JointID.Head].Position;
            KinectID = user.TrackingID;
            break;
        }
}
```

这样，我们就将原始的头部坐标记录到了Head变量中，以供下一步处理。

### 11.3.3　根据Kinect和屏幕的位置关系转换坐标

目前得到的头部位置是Kinect坐标系中的位置，因此需要将其转换为屏幕坐标系的坐标，如图11-3所示。




图11-3　头部坐标变换

全息显示的调整过程需要精确的相对位移，而绝对坐标只用来调整虚拟物体的初始位置。因此并不需要进行过于精确的坐标系标定过程，而是简单地采集Kinect和屏幕的相对距离以及旋转方向，然后输入到程序中进行坐标变换即可。这里我们仅考虑左旋、右旋、上仰以及位移四种坐标变换方式。在Kinect组件类中添加如下代码：

```
public static Vector3 KinectOffset = new Vector3(0, 0, 0);
public static bool LeftRotateBool = false;
```

```
public static bool RightRotateBool = false;
public static bool UpsideBool = false;

public void LeftRotate(Vector3 kinectVector)
{
    Vector head0;
    head0 = head;
    head.X = head0.Z + kinectVector.X;
    head.Y = head0.Y + kinectVector.Y;
    head.Z = -head0.X + kinectVector.Z;
}

public void RightRotate(Vector3 kinectVector)
{
    Vector head0;
    head0 = head;
    head.X = -head0.Z + kinectVector.X;
    head.Y = head0.Y + kinectVector.Y;
    head.Z = head0.X + kinectVector.Z;
}

public void Upside(Vector3 kinectVector)
{
    Vector head0;
    head0 = head;
    head.X = head0.X + kinectVector.X;
    head.Y = -head0.Y + kinectVector.Y;
    head.Z = head0.Z + kinectVector.Z;
}

public void Translation(Vector3 kinectVector)
{
    head.X += kinectVector.X;
    head.Y += kinectVector.Y;
    head.Z += kinectVector.Z;
}
```

其中，LeftRotate、RightRotate和Upside函数分别为左旋、右旋和上下倒转三种旋转方式，在旋转之后还要加上Kinect和屏幕的相对位移矢量kinectVector。Translation函数则为不加旋转的平移。

在得到Head坐标后，调用上述函数进行坐标变换：

```
if (LeftRotateBool)
    LeftRotate(KinectOffset);
else if (RightRotateBool)
    RightRotate(KinectOffset);
else if (UpsideBool)
    Upside(KinectOffset);
else
    Translation(KinectOffset);
```

至此，我们就完成了Kinect组件类，并得到了头部相对于屏幕的坐标。

# 11.4　三维图形引擎

在获取了头部坐标之后，剩下的就只有三维图形的绘制工作了。这本身是一项比较复杂的工程，有很多三维图形引擎和游戏引擎可以用来辅助开发，该项目是采用XNA游戏框架来完成的。由于这并不是本书讲述的重点，因此这里只通过实现XNA中一个绘制可见模型类的代码，来简单讲解三维模型的绘制过程，为下一节的视角变换做铺垫。

## 11.4.1　创建可见模型绘制类

在Visual Studio 2010中新建一个C#类，命名为“IllModel.cs”，用来实现模型绘制功能，如图11-4所示。

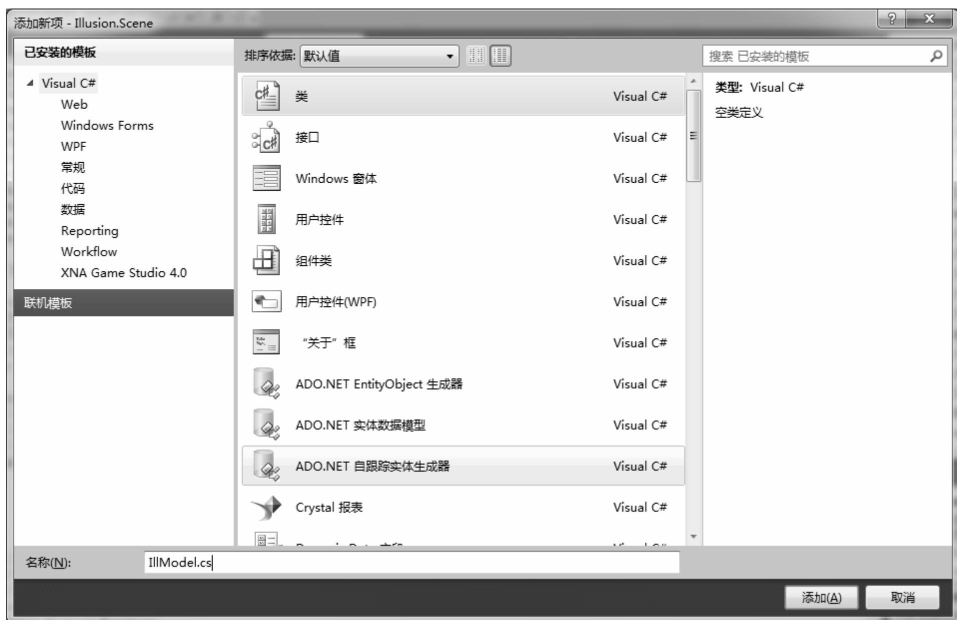



图11-4　创建一个可见模型绘制类

## 11.4.2　构建模型世界矩阵

一个3D模型需要经过世界矩阵、视图矩阵和投影矩阵的转换，再经过裁剪等一系列的处理才能转换为屏幕坐标并绘制出来。其中视图矩阵和投影矩阵都和观察点的坐标有关，而世界矩阵仅仅和模型本身有关。因此，模型绘制的第一步就是计算其世界矩阵。相关代码如下：

```
public class IllModel
{
    string modelAssetName;
```

```
public string ModelAssetName
{
    get { return modelAssetName; }
    set { modelAssetName = value; }
}

Vector3 position;
public Vector3 Position
{
    get { return position; }
    set { position = value; }
}

Vector3 scale = new Vector3(1.0f, 1.0f, 1.0f);
public Vector3 Scale
{
    get { return scale; }
    set { scale = value; }
}

Vector3 velocity;
public Vector3 Velocity
{
    get { return velocity; }
    set { velocity = value; }
}

Vector3 yawPitchRaw = new Vector3(0,0,0);

private Game game;

public Matrix Rotation
{
    get
    {
        return Matrix.CreateFromYawPitchRoll(yawPitchRaw.X,
                                             yawPitchRaw.Y,
                                             yawPitchRaw.Z);
    }
}

public Matrix World
{
    get
    {
        Matrix mt1 = Matrix.CreateScale(scale);
        Matrix mt2 = Rotation;
        Matrix mt3 = Matrix.CreateTranslation(position);
        return mt1 * mt2 * mt3;
    }
}

public IllModel(string model, Vector3 position, Vector3 yawPitchRaw,
```

```
                        float scale, Game game)
        {
            this.position = position;
            this.yawPitchRaw = yawPitchRaw;
            this.scale = new Vector3(scale, scale, scale);
            this.modelAssetName = model;
            this.game = game;
        }
    }
```

世界矩阵代表着模型在三维空间中的位置参数，包括位移、旋转和放缩。而上述代码中的`position`、`yawPitchRaw`和`scale`分别代表了这三个参数，其中参数`yawPitchRaw`表示模型在yaw、pitch、raw三个轴上的旋转程度。

但是仅仅有参数是不够的，还需要将其转换为矩阵。这里只需调用XNA中相应的接口来创建矩阵，最后再通过矩阵乘法合并成模型的世界矩阵：

```
public Matrix World
{
    get
    {
        Matrix mt1 = Matrix.CreateScale(scale);
        Matrix mt2 = Rotation;
        Matrix mt3 = Matrix.CreateTranslation(position);
        return mt1 * mt2 * mt3;
    }
}
```

这样就得到了该模型的世界矩阵`World`，下一步就是进行绘制工作了。

### 11.4.3　绘制模型

上面提到了模型绘制需要三个矩阵，我们已经得到了世界矩阵，因此需要将视图矩阵和投影矩阵作为参数传入绘制函数。代码如下：

```
public void Draw(GameTime gameTime, Matrix view, Matrix projection)
{
    Model model = game.Content.Load<Model>(modelAssetName);
    Matrix[] transforms = new Matrix[model.Bones.Count];

    model.CopyAbsoluteBoneTransformsTo(transforms);
    game.GraphicsDevice.BlendState = BlendState.AlphaBlend;
    game.GraphicsDevice.DepthStencilState = DepthStencilState.Default;

    foreach (ModelMesh mesh in model.Meshes)
    {
        foreach (BasicEffect effect in mesh.Effects)
        {
            effect.EnableDefaultLighting();
            effect.World = transforms[mesh.ParentBone.Index] * World;
            effect.View = view;
            effect.Projection = projection;
```

```
            }
            mesh.Draw();
        }
    }
```

在分析这段代码之前，我们先来了解模型数据的组织方式：XNA用Model保存模型数据，其中包含一些“小”物体，称之为网格，ModelMesh类表示这些网格数据。同样地，网格也由好几部分构成，每个部分都有其颜色、纹理和光照等绘制参数，其中参数相同的部分称之为效果，在XNA中用Effect类记录。在XNA框架下，如果对每一个Effect都设置了相应参数和三矩阵，绘制模块就可以自动完成模型的绘制工作。

上述代码是一段经典的XNA模型绘制代码，首先载入模型数据并调整绘图设备的相应参数。接着foreach循环遍历模型的每个网格及其中的每一个Effect对象，应用光照的默认参数，设置被绘制对象的世界、视图和投影矩阵。最后调用网格绘制函数就可以自动绘制出整个模型了。

到目前为止，我们已经完成了三维模型的绘制工作。再加上第一步获取的头部坐标，下面只要将这两者结合在一起就可以了。

## 11.5　根据头部位置更新绘制图像

在分析如何利用头部坐标更新绘制图像之前，我们先来了解一下三维图像的绘制原理，如图11-5所示。

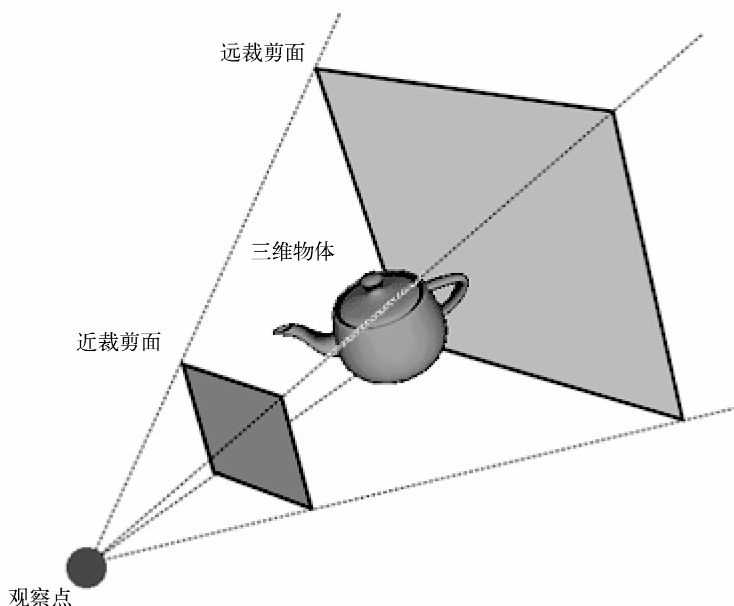



图11-5　三维图像绘制原理

如前文所述，三维绘制的三个核心矩阵为世界矩阵、视图矩阵和投影矩阵。其中，世界矩阵由图中三维物体的位置、大小、旋转等参数构成，可以将其理解为“决定物体在哪里”的数据；视图矩阵由观察点的位置和朝向构成，也就是“在哪里、向哪里看”的参数；最后视锥体的大小、远近裁剪面的数据则表示了“屏幕有多大，朝向哪里”。所以，知道了“物体在哪里”以及“向哪里看”，又知道了“屏幕有多大，朝向哪里”，就能在屏幕上得到绘制图像了。

由此可见，想要根据头部位置来更改显示的图像，首先要修改视图矩阵；此外，由于屏幕的朝向参数是相对于观察点位置的，观察点改变了，投影矩阵也要随之改变。本节的代码是建立在项目的主干流程中的，我们将重点讨论如何实现这两个矩阵的数据更新。

## 11.5.1 修改视图矩阵

首先更新视图矩阵的数据，其代码如下：

```
KinectComponent kinect;
public static bool kinectbool = true;
Vector3 pos = new Vector3(0, 0, 0.4f);
Vector3 target;
Vector3 up;
Matrix view, projection;

protected override void Draw(GameTime gameTime)
{
    if (kinectbool && kinect != null)
    {
        pos = kinect.Headposition;
        target = pos + new Vector3(0, 0, -3f);
    }
    view = Matrix.CreateLookAt(pos, target, up);
}
```

其中pos、target和up分别表示观察者的位置、视线目标和正方向，View表示视图矩阵，projection表示投影矩阵。当Kinect捕捉到头部位置时，应该更新观察者的位置，但是视线的方向并不会发生改变。因为视线方向用于确定屏幕平面，投影矩阵用于确定视锥体，所以视线方向向量应该永远垂直于屏幕所在的平面，而不是随着观察者位置发生改变。如图11-6所示。

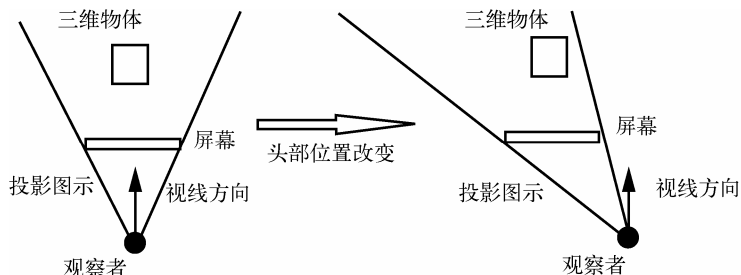



图11-6 头部位置改变而视线方向不变

因此，视线目标只需要根据观察者的位置进行平移即可：

```
target = pos + new Vector3(0, 0, -3f);
```

最后，调用XNA中的`CreateLookAt`方法计算出视图矩阵`view`。

## 11.5.2　修改投影矩阵

与视图矩阵相比，投影矩阵的更新要复杂一些。这是因为常用的投影矩阵都是基于正视锥体的，即视线方向通过视锥体的中心线。而我们要使用的可能是斜投影矩阵，由于很多图像引擎都不提供斜投影矩阵的构造方法，这里将介绍一种将正投影矩阵转化为斜投影矩阵的方法。

**1. 将正投影矩阵转化为斜投影矩阵**

世界矩阵、视图矩阵和投影矩阵决定了一个模型在屏幕上绘制出的图像，而这三个矩阵的计算是有顺序的。首先通过世界矩阵将模型变换到统一的世界坐标系中，再通过视图矩阵将其转换到视点空间中，最后通过投影矩阵将前面得到的顶点坐标转化为一个正方体内的坐标。

如图11-7所示，经过视图矩阵的转化，空间的顶点坐标被转换为观察者坐标系的坐标，黑色实线代表当前阶段的坐标轴，图中的实心点代表视野空间。通过正投影矩阵的变换，视锥体中的梯台区域被转化为正立方体，顶点$(l, b, n)$和$(r, t, m)$分别被转化为$(l', b', n')$和$(r', t', m')$。

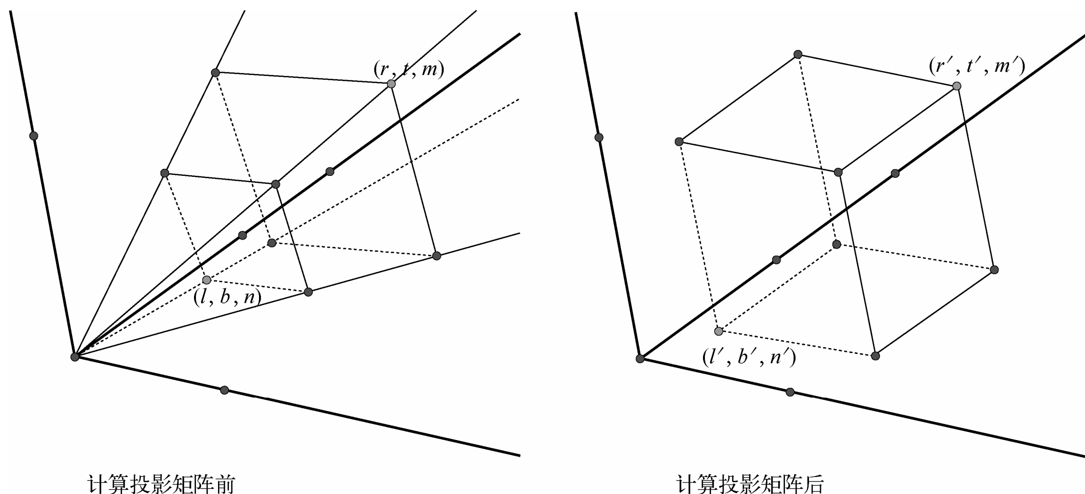



图11-7　投影矩阵功能

实现斜投影矩阵最简便的方法是，在投影矩阵之前加入一个新矩阵，并将视锥体内的顶点都转换到正视锥体中，这样不会影响后面的运算。通过这种矩阵变换，对每个与屏幕平行的平面上的所有点进行平移，将以观察方向为中心的点平移到以视线方向为中心就可以解决问题了，如图11-8所示。

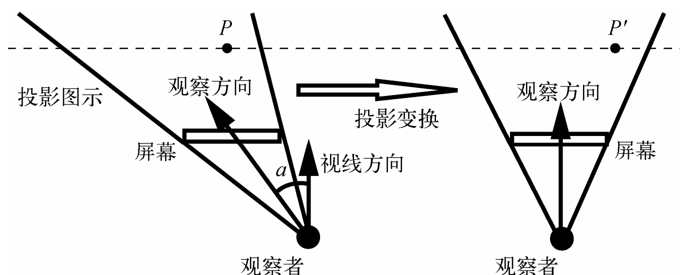



图11-8 新加一个矩阵实现斜投影矩阵

通过这个中间矩阵将绿色点$P$转换为$P'$，实现斜投影矩阵向正投影矩阵的转换。令观察方向的单位向量为***Direction***，视线方向的单位向量为***Lookat***、观察方向与视线方向的夹角为$a$。不难得出，每个对应点的坐标应该有如下的对应关系:

$$Pos' = Pos + \left[(-Pos.Z)\times \boldsymbol{Lookat} - \frac{(-Pos.Z)}{\cos a}\times \boldsymbol{Direction}\right]$$

所以中间矩阵***M***应符合如下关系：

$$\boldsymbol{M}\begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} = \begin{pmatrix} x + z\times\dfrac{1}{\cos a}\times direction.X \\ Y + z\times\dfrac{1}{\cos a}\times direction.Y \\ z \\ w \end{pmatrix}$$

因此在视图矩阵和正投影矩阵之间再乘上一个如下的矩阵，就可以得到符合要求的斜投影矩阵了：

$$\begin{bmatrix} 1 & 0 & \dfrac{direction.X}{\cos a} & 0 \\ 0 & 1 & \dfrac{direction.Y}{\cos a} & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

代码实现如下：

```
public static Matrix CreateIlluMatrix(Vector3 direction)
{
    Matrix m = Matrix.Identity;
    float cosa = Vector3.Dot(Vector3.Forward, direction);
    if (cosa!=0)
    {
        m.M31 = direction.X / cosa;
        m.M32 = direction.Y / cosa;
    }
```

```
    return m;
}

illuMatrix = CreateIlluMatrix(Vector3.Normalize(Vector3.Zero - pos));
float a = 0.01f / pos.Z;
projection = illuMatrix * Matrix.CreatePerspective(ScreenWidth * a,
                                                   ScreenWidth / AspectRatio * a,
                                                   0.01f, 7000f);
```

其中，CreateIlluMatrix函数为计算中间矩阵***M*** 的函数，最后将中间矩阵和正投影矩阵相乘得到所需的斜投影矩阵projection。

2. 利用XNA提供的函数生成斜投影矩阵

由于很多图像引擎只提供了正投影矩阵的生成接口，因此上面介绍的斜投影矩阵生成方法适用于所有的图像引擎。但是在XNA框架下，其本身提供了一个生成斜投影矩阵的方法，传入相应的参数就可以得到斜投影矩阵：

```
public static Matrix CreatePerspectiveOffCenter (
    float left,
    float right,
    float bottom,
    float top,
    float nearPlaneDistance,
    float farPlaneDistance
)
```

其中，left、right、bottom、top分别表示视锥体在近截平面上的左、右、上、下边界值，nearPlaneDistance和farPlaneDistance表示近、远截平面的距离。由此，我们可以利用现有的数据来计算斜投影矩阵，代码如下：

```
public static Matrix CreateXNAProjection(Vector3 direction,
                                         float screendepth,
                                         float screenwidth,
                                         float aspectRatio,
                                         float nearPlaneDistance,
                                         float farPlaneDistance)
{
    float cosa = Vector3.Dot(Vector3.Forward, direction);
    Vector3 screenCenter = direction * screendepth / cosa;
    float a = nearPlaneDistance / screendepth;
    float left = (screenCenter.X - screenwidth / 2) * a;
    float right = (screenCenter.X + screenwidth / 2) * a;
    float screenHeight = screenwidth / aspectRatio;
    float top = (screenCenter.Y + screenHeight / 2) * a;
    float bottom = (screenCenter.Y - screenHeight / 2) * a;
    return Matrix.CreatePerspectiveOffCenter(left, right, bottom, top,
                                             nearPlaneDistance,
                                             farPlaneDistance);
}

projection = CreateXNAProjection(Vector3.Normalize(Vector3.Zero - pos),
                                 pos.Z,
```

```
                    ScreenWidth,
                    AspectRatio,
                    0.01f,
                    7000f);
```

这里的CreateXNAProjection函数就是封装好的斜投影矩阵计算函数。其中，screenCenter计算的是屏幕的中心坐标，screenwidth和screenheight表示屏幕的宽和高。最终将计算出的left、right、top、bottom传入CreatePerspectiveOffCenter函数即可。

至此，我们就完成了全息显示项目的全部核心代码，包括捕捉头部位置、绘制三维模型以及改变矩阵调整显示图像。最终，程序的效果如图11-9所示。



观察者在屏幕右侧看到的画面



观察者在屏幕下方看到的画面

图11-9　最终运行结果

## 11.6　小结

通过以上的项目实例，我们了解了一个完整的Kinect三维应用是如何一步步开发出来的。这个项目的优点并不仅限于其独特的全息显示方式，将之应用到文物保护也是值得读者借鉴的。

此外，它的应用前景也很好，因为只要是有三维显示的地方，都可以用到Kinect全息显示。例如，在该方案的支持下，游戏、动画或电影可以更加真实，而不只是在一块平面上的投影，3D游戏可以具备更强的代入感；一块配有Kinect和屏幕的小型广告牌采用该方案后，无论大家是步行还是驾车，是等人还是散步，广告都会随时移动，有身临其境的画面感，增加趣味性的同时也极大地提高了广告感染力，从而带来经济效益；机械设计师可以轻松地通过移动头部观察其正在设计的物件的样貌，解决繁杂的移动勘测问题，从而幅提高工作效率……

同时，该方案还可以和传统的3D显示技术结合，直接追踪观察者左眼和右眼相对屏幕的位置，并计算出左眼和右眼不同的投影，从而提供更逼真的表现力。此外，该方案也可与基于Kinect的远距离多点触控技术相结合，使计算机软件用户界面不再集中在一个平面上，而是转变为不同朝向、多层次的三维用户界面。

第 12 章

# 基于Kinect的自主移动机器人的设计与实现

前面几章介绍的Kinect实战项目都有一个共性：以Kinect设备作为人机交互的工具，识别使用者的动作并做出不同的响应，如果给这种输入方式加上一个方向的话，应该属于向内输入。我们不妨做一些改变，把输入方向反过来，即让Kinect去观测外界的环境并做出反馈，就像人的眼睛一样。由此，可以将Kinect作为机器人的视觉传感器，这是完全不同于之前项目的一种全新的Kinect应用方式。本章就来介绍一个以这种方式实现的研发项目——Kinect自主移动机器人（KRobot），如图12-1所示。

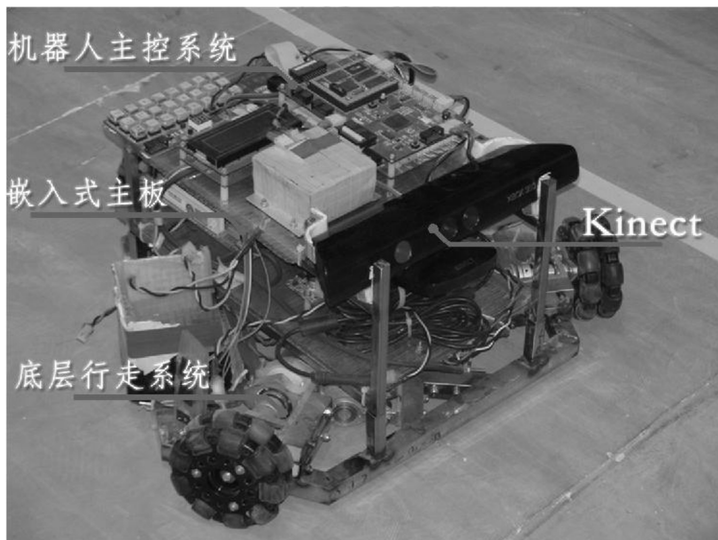



图12-1　KRobot机器人

该项目入围了微软亚洲研究院举办的“2012微软精英大挑战”决赛，并最终获得第一名的优异成绩。它不仅将Kinect创新性地应用到了机器人领域，而且还利用Kinect的数据实现了极佳的效果，这是非常值得读者学习和借鉴的。

## 12.1　KRobot 项目简介

通常，传统机器人的避障功能会利用机器视觉技术来识别机器人附近的障碍物，并做出响应。其中，充当机器人“眼睛”的部分，一般是RGB摄像头、红外线摄像头或激光收发器等传感器。但是这些传感器都有其局限性：RGB摄像头虽然可以获得全视野的图像信息，但其识别算法基于彩色位图的图像识别位图，受光线的影响较大，实际应用性较差；红外线摄像头和激光收发器虽然可以获取可靠的数据，但是只能获得某一方向上的障碍信息。而Kinect传感器不仅具备传统RGB摄像头的功能，还拥有稳定且视野宽广的深度摄像机，以及大角度的麦克风阵列。这不仅覆盖了现有传感器的功能，还提高了数据的密度和可靠性。因此，KRobot项目以Kinect取代现有的传感器，来完成机器视觉识别，并获取障碍物的信息，进而控制机器人完成完整的避障动作。

不仅如此，KRobot还可以完成一些更加系统且实用的任务。比如，对视野内的目标人物进行持续追踪，保持一定距离并跟随；根据语音命令的发出方向进行转向。该项目已经实现了这两种动作，即人体跟踪和声源定位。

同时，该项目还借助Kinect强大的人机交互功能，实现了人与机器的自然交互。使用者可以通过直观的动作来完成机器人的操控任务。

## 12.2　技术实现概述

本项目的技术架构分为三大部分，分别是视觉平台、主控系统以及底层行走机构。视觉平台不仅需要采集Kinect的信息，还需要根据已知信息分析外界当前环境，并做出下一步行动决策。这相当于人的大脑，而Kinect就相当于人的眼睛与耳朵。主控系统负责接收视觉平台的指令，采集其他传感器的信息并且控制电机做出相应动作。这相当于人的神经中枢，控制身体各部位执行大脑的指令，有承上启下的作用。而底层的行走机构包含了直流电机、电池以及金属车架，相当于人的骨骼与肌肉。只有这三部分紧密协作，机器人才能正常运转。

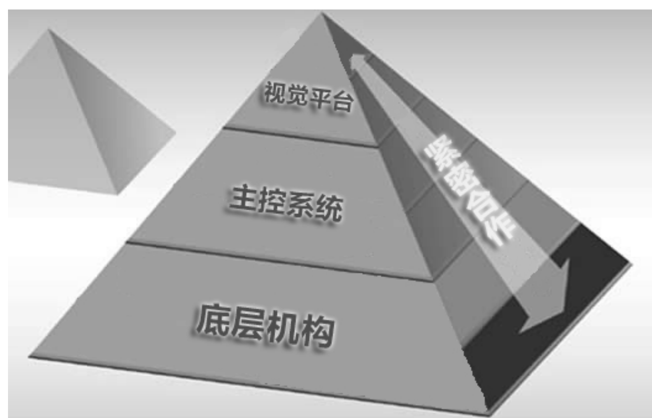



图12-2　KRobot的三层技术实现

项目中的Kinect部分，需要完成四个功能：障碍规避、人体跟踪、声源定位和自然交互。

- 障碍规避需要先完成摄像机标定，再依靠深度摄像头获取的原始数据流完成图像识别，最终得到障碍的信息；
- 人体跟踪通过Kinect SDK提供的骨骼跟踪功能完成；
- 声源定位通过Kinect的麦克风阵列数据获取声源方向；
- 自然交互和其他Kinect应用的交互方式类似，通过识别人体动作来实现交互功能。

在后面几节中，我们将分析视觉平台和主控平台中与Kinect相关的每一项技术实现，这些代码使用的编程语言是C++。读者在阅读本章的代码之前，需要先学习Kinect SDK文档中关于C++语言的内容。本项目采用的是1.0版本的Kinect for Windows SDK，针对后续SDK的版本升级，请参考相关文档和程序。

由于该项目的底层机构过于复杂，而且和本书的主题相距较远，这里不做介绍，感兴趣的读者可以参阅相关书籍。

## 12.3　利用深度数据进行摄像机标定

在进行摄像机标定之前，先来思考这样一个问题：我们定义了如图12-3所示的坐标系，假设Kinect放置在$x$轴上，其中心位置对应于坐标原点，通过前面的章节可知，Kinect可以得到深度图上某一点在Z方向的深度距离，那么我们可以得到这个点在三维空间中$x$轴和$y$轴的坐标吗？

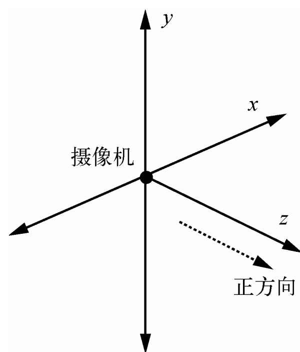



图12-3　摄像机空间示意图

从另一个角度来看，我们需要知道采集到的深度图像的像素点与真实世界物理坐标系之间的联系，并得到一种二维到三维的映射关系。想要得到这种映射关系，就需要进行摄像机标定。简单地说，这就是对近似描述整个摄像头成像过程的数学模型的各个参数进行计算的过程。通过摄像机标定，只要给定实际空间中某一点的三维坐标，我们就可以通过计算得到该点在摄像头成像平面上对应的二维坐标。

之前我们已经提到，SDK包含了相应的转换函数，但在这里我们并不使用它，而是利用成像原理完成标定的计算过程。这有助于读者深入理解Kinect深度摄像头的成像原理。

摄像机标定过程比较复杂，首先需要确定成像模型。在实际项目的制作当中，我们发现光学中理想的针孔模型与深度摄像头的成像模型极为相似。在精度要求不高的情况下，可以很容易地计算出X和Y方向上的距离，具体思路如下。

X和Y方向上的距离是控制机器人运动的，这是一个相对距离，在二维的深度图像中，可以通过像素点$(u, v)$偏离图像中心位置$(u_0, v_0)$的坐标利用针孔成像模型计算得到。

$$\frac{d_x}{|u-u_0|}=\frac{depth(u,v)}{f_X}$$

$$\frac{d_y}{|v-v_0|}=\frac{depth(u,v)}{f_Y} \quad (12\text{-}1)$$

上式中，$(d_x, d_y)$表示像素点$(u, v)$相对中心位置$(u_0, v_0)$在X和Y方向上的偏移距离，$depth(u, v)$表示该点对应的深度距离，$f_X$ 和$f_Y$ 为摄像机的内部参数，表示X和Y方向的焦距，可假设为一个定值。

而通过像素点$(u, v)$和深度摄像头的内部参数，可以将机器人运动距离的计算简化为：

$$d_x=|u-u_0|\frac{depth(u,v)}{f}$$

$$d_y=|v-v_0|=\frac{depth(u,v)}{f} \quad (12\text{-}2)$$

$$z=depth(u,v)$$

SDK提供了一个名为NUI_CAMERA_DEPTH_NOMINAL_INVERSE_FOCAL_LENGTH_IN_PIXELS的宏，代表的是深度摄像焦距值的倒数，单位是像素值。将该宏代入上式进行计算，相关代码如下：

```
#define Inv_fd NUI_CAMERA_DEPTH_NOMINAL_INVERSE_FOCAL_LENGTH_IN_PIXELS
float edge_left = (float)fabs(depth*(160-left_edge_point.x)*Inv_fd);
```

将深度摄像头采集的图像尺寸设置为320×240，图像中心位置即为(160, 120)。由此，利用深度图像就可以得到控制机器人运动所需的三维距离信息了。

## 12.4　利用深度数据实现障碍规避

避障的前提是识别前方的障碍物，构成障碍物的条件只与机器人的大小以及物体的位置有关，不需考虑物体的形状和颜色，所有阻碍机器人前进路线的物体都可以被视为障碍物。因此利用深度数据流可以很简单地进行处理，而利用传统的彩色摄像头，则需要预先了解障碍物的信息才能达到很好的识别效果。上一节已经介绍了利用深度图像数据获取机器人所需运动数据的方法。如果得到了某个像素的二维坐标，我们就可以控制机器人运动到该像素坐标所对应的物理位置；而如果得到的是障碍物的边缘像素坐标，那么就必须让机器人避开这个障碍物。实际上，机器人只运动在X-Z平面，因此我们只需要关注式(12-2)中的$d_x$和$z$即可。本节将详细讲解如何利用深度数据实现避障功能。深度摄像机的深度值示意图如图12-4所示。

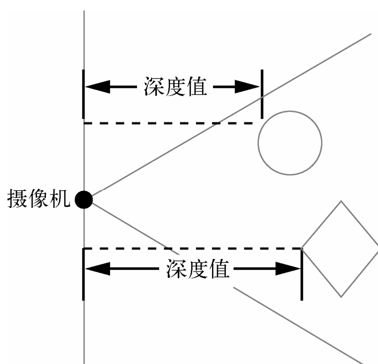



图12-4　深度摄像机深度值示意图

如前文所述，Kinect得到的深度数据是一个深度距离的二维数组，单位是毫米。系统可以利用深度数组构造出一个灰度图像，在得到坐标值后，返回深度数组查找需要的深度信息，并进行偏移距离的计算。该处理过程具体如下。

首先获取深度数据流，由于Kinect深度传感器的探测范围约为0.8 ~ 3.5米，因此可设置阈值（3米）去除过远的距离。然后复制实际的深度数据缓冲区和转化后的深度图像数据缓冲区（255归一化），为了去除地面的影响（假设空旷无障碍物和有障碍物的深度图地面是一样的），这里将无障碍物的情况作为参考帧对后续帧进行帧差法。接下来，对图像进行膨胀腐蚀以尽量去除噪声的影响。然后对得到的图像进行连通体识别和轮廓面积计算，得到符合一定规则的连通体（比如程序中的设定面积大于1000），再对得到的连通体进行障碍物判断，并得到最终认可的障碍物。接着利用障碍物的坐标计算障碍物与机器人之间的距离，并根据一定的几何关系计算机器人避障所需的偏转角度以及运动参数。

考虑到开发的需要，我们应在SDK的基础上借助第三方开源视觉库OpenCV，完成Kinect机器人视觉平台的开发工作。它实现了图像处理和计算机视觉方面的很多通用算法，非常实用。如果读者此前并未接触过这个库，可以参考OpenCV中文网（http://www.opencv.org.cn）进行学习。

## 12.4.1　获取彩色图和深度图数据

参考SDK中提供的C++实例程序，可以完成Kinect彩色图和深度图的获取，具体代码如下：

```
//设置参数为深度数据（含player index）、彩色数据、骨架数据以及声音数据
DWORD nuiFlags = NUI_INITIALIZE_FLAG_USES_DEPTH_AND_PLAYER_INDEX |
                 NUI_INITIALIZE_FLAG_USES_COLOR |
                 NUI_INITIALIZE_FLAG_USES_SKELETON |
                 NUI_INITIALIZE_FLAG_USES_AUDIO;
hr = m_pNuiSensor->NuiInitialize(nuiFlags);
if (E_NUI_SKELETAL_ENGINE_BUSY == hr)
{
    nuiFlags = NUI_INITIALIZE_FLAG_USES_DEPTH |
               NUI_INITIALIZE_FLAG_USES_COLOR |
               NUI_INITIALIZE_FLAG_USES_AUDIO;
```

```
    hr = m_pNuiSensor->NuiInitialize(nuiFlags) ;
}

if (hr != S_OK)
{
    cout << "NuiInitialize failed" << endl;
    return hr;
}

//骨架事件
m_hNextSkeletonEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
if (HasSkeletalEngine(m_pNuiSensor))
{
    hr = m_pNuiSensor->NuiSkeletonTrackingEnable(m_hNextSkeletonEvent, 0);
    if (FAILED(hr))
    {
        cout << "Could not skeleton" << endl;
        return hr;
    }
}

//彩色图事件，彩色图尺寸为640*480
m_hNextVideoFrameEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
m_pVideoStreamHandle = NULL;
hr = m_pNuiSensor->NuiImageStreamOpen(
         NUI_IMAGE_TYPE_COLOR,
         NUI_IMAGE_RESOLUTION_640x480,
         0,
         2,
         m_hNextVideoFrameEvent,
         &m_pVideoStreamHandle);
if (FAILED(hr))
{
    cout << "Video Frame open failed!\n" << endl;
    return hr;
}

//深度图事件，深度图尺寸为320*240
m_hNextDepthFrameEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
m_pDepthStreamHandle = NULL;
hr = m_pNuiSensor->NuiImageStreamOpen(
    NUI_IMAGE_TYPE_DEPTH_AND_PLAYER_INDEX,
    NUI_IMAGE_RESOLUTION_320x240,
    0,
    2,
    m_hNextDepthFrameEvent,
    &m_pDepthStreamHandle);
if (FAILED(hr))
{
    cout << "Could not open depth stream video" << endl;
    return hr;
}
```

然后定义OpenCV的标准图像类型`IplImage`，存取彩色图和深度图，相关代码如下：

```
VideoFrame = cvCreateImage(cvSize(COLOR_WIDTH, COLOR_HEIGHT), IPL_DEPTH_8U, 4);
m_DepthFrame = cvCreateImage(cvSize(DEPTH_WIDTH, DEPTH_HEIGHT), IPL_DEPTH_8U, 1);
```

## 12.4.2　处理深度图和深度数据

根据前文提供的思路，将深度数据作为灰度图进行处理，相应的视觉算法通过OpenCV库来完成。

首先定义一个深度图视觉算法处理函数：

```
void CKRobotVisionDlg::Robo_DepthVision(IplImage* src, IplImage* dst)
{
    //复制原始深度图
    IplImage* depth_copy = cvCreateImage(cvSize(src->width, src->height),
                                            IPL_DEPTH_8U, 1);
    IplImage* depth = cvCreateImage(cvSize(src->width, src->height),
                                    IPL_DEPTH_8U, 1);

    //进行阈值操作后的深度图
    IplImage* depth_Morphic_Binary = cvCreateImage(cvSize(src->width, src->height),
                                                    IPL_DEPTH_8U, 1);

    //进行形态学操作后的深度图
    IplImage* depth_Morphic = cvCreateImage(cvSize(src->width, src->height),
                                            IPL_DEPTH_8U, 1);

    //进行边界链码提取后的深度图
    IplImage* depth_Morphic_Contour = cvCreateImage(cvSize(src->width, src->height),
                                                    IPL_DEPTH_8U, 1);

    //OpenCV中进行边界提取定义的变量
    CvMemStorage* stor;
    CvSeq* cont_first;

    //连通体面积
    long S_max = 0;
}
```

在这段代码里，`cvCreateImage`是OpenCV中创建图像数据头部并分配数据的函数，即`IplImage`对象初始化，其中要用到的数据如下。

- `depth_copy`：复制原始深度图。
- `depth`：原始深度图。
- `depth_Morphic_Binary`：进行阈值操作后的深度图。
- `depth_Morphic`：进行形态学操作后的深度图。
- `depth_Morphic_Contour`：进行边界链码提取后的深度图。

接下来完成具体的处理工作，具体代码如下：

```
//将深度帧与参考帧进行差帧，消除地面的影响
Robo_PreDepthImage(depth_copy, depth);
//单一阈值分割
cvThreshold(depth,depth_Morphic_Binary, 60, 255, CV_THRESH_TOZERO);
//腐蚀运算
cvErode(depth_Morphic_Binary, depth_Morphic_Binary, NULL, 1);
//膨胀运算
cvDilate(depth_Morphic_Binary, depth_Morphic, NULL, 1);
```

```
//进行复制
cvCopy(depth_Morphic, depth_Morphic_Contour);
//提取图中的边界，这是OPENCV中的轮廓提取函数
cvFindContours(depth_Morphic_Contour, stor, &cont_first, sizeof(CvContour),
               CV_RETR_LIST, CV_CHAIN_APPROX_SIMPLE, cvPoint(0,0));
//该数组存放连通体的顶点坐标、长和宽，以及旋转参数。
CvBox2D cont_box[10];
for (; cont_first; cont_first = cont_first->h_next)
{
    S_max = (long)fabs(cvContourArea(cont_first, CV_WHOLE_SEQ));
    if (S_max > 10 * 100)
    {
        cont_box[num++] = cvMinAreaRect2(cont_first, NULL);
    }
}
```

首先对深度图进行边缘处理，提取图像的边界。其中使用了大量OpenCV中的算法函数，如下所示。

- `cvThreshold`：单一阈值分割。
- `cvErode`：腐蚀运算。
- `cvDilate`：膨胀运算。
- `cvCopy`：进行复制。
- `cvFindContours`：提取图中的边界。

接下来，将其中面积大于1000的连通体存入`CvBox2D`类型数组中，就可以提取出前方障碍物的轮廓和坐标位置了。

最后将其绘制出来，效果如图12-5所示。

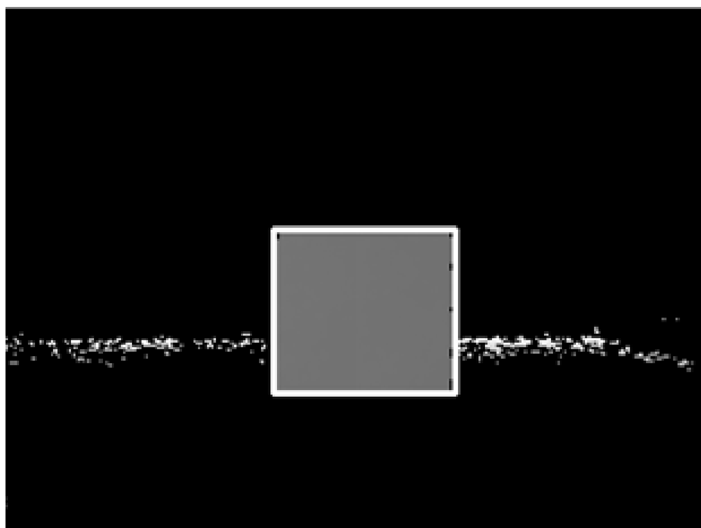


图12-5　障碍物提取结果

### 12.4.3 制定障碍物判定规则

我们可以通过深度图像处理函数得到机器人前方物体的轮廓。但是对机器人运动来说，前方物体并不一定构成阻碍，这与机器人的大小以及高度有关，因此需要制定具体的障碍物判定规则。

首先声明一个障碍物类以便后续处理，相关代码如下：

```
typedef struct
{
    int LorRTag;    //障碍物相对于机器人的位置，0-左边，1-中间，2-右边
    bool active;    //障碍物状态

    float kRobot_l1;    //机器人左半边宽度
    float kRobot_l2;    //机器人右半边宽度
    float Obj_width;    //障碍物宽度
    float edge_dis;     //机器人向一个方向移动的距离
    float other_edge_dis;   //机器人向另一个方向移动的距离

    CvBox2D ObjBox;     //障碍物构成的二维盒子

    //发送给机器人的相关数据
    int TurnLorR;   //机器人偏转方向
    float Left_edge;    //障碍物左边缘距机器人中心的距离
    float Right_edge;   //障碍物右边缘距机器人中心的距离
    float edge_depth;   //边缘点对应的深度距离

} Robo_Object_Array;

Robo_Object_Array robo_object[3];   //三种障碍物分布情况
```

这个类不仅用于记录障碍物参数，还用于主控部分对机器人避障动作的控制。然后进行障碍物的筛选，相关代码如下：

```
for (int i=0; i<num; i++)
{
    //Robo_CheckObj是障碍物判定函数
    if (Robo_CheckObj(&temp_obj, cont_box[i]))
    {
        robo_object[temp_obj.LorRTag] = temp_obj;
        robo_object[temp_obj.LorRTag].theta =
            Robo_CalRoboTheta(&robo_object[temp_obj.LorRTag]);
        robo_object[temp_obj.LorRTag].active = true;    //标志位置位
        Robo_Obj_Num++;     //障碍物数目加1
        rvDrawBox(robo_object[temp_obj.LorRTag].ObjBox,
                  depth_process,
                  CV_RGB(255, 255, 255));  //在图上绘出障碍物
    }
}
```

遍历连通体，根据障碍物判定规则进行判别。首先将物体相对于机器人的方位分为左边、中间以及右边三种情况，然后分别求得物体的左、右边缘距离以及中间距离（这种情况下取左、右边缘距离中较小的那个）。边缘距离指的是边缘位置相对机器人中心位置在X方向上的相对距离，

为了减小边缘数据跳变，求解时通常要在一定的边缘像素范围内进行统计平均。对上述三种情况下的边缘距离进行判定，距离大于机器人的宽度表示不对机器人构成威胁，即该物体不是障碍物，否则将该物体视为障碍物，并将其赋给robo_object数组。

核心的障碍物判断函数Robo_CheckObj代码如下：

```
BOOL CKRobotVisionDlg::Robo_CheckObj(Robo_Object_Array* robo_obj, CvBox2D box)
{
    BOOL obj_status = FALSE;

    /*找到二维盒子的左、右边缘并转换为Kinect视角的三维坐标，代码略*/

    //左、右两个盒子的距离足够大时，不对机器人构成威胁
    if (fabs(depth_left - depth_Right) < 400)
    {
        //考虑KRobot的死区
        depth_left = depth_left ;
        depth_Right = depth_Right ;

        float edge_left = (float)fabs(depth_left *
                              (160 - left_edge_point.x) * Inv_fd);
        float edge_right = (float)fabs(depth_Right *
                              (right_edge_point.x - 160) * Inv_fd);

        //分别处理左、中、右三种情况
        if (bottom.y > 150)
        {
            if (left_edge_point.x < 160 && right_edge_point.x <= 160)
            {
                if (edge_right < 500)
                {
                    robo_obj->LorRTag = 2;
                    robo_obj->edge_depth = depth_Right + KRobot_length;

                    robo_obj->ObjBox = box;
                    robo_obj->Left_edge = edge_right;
                    robo_obj->Right_edge = edge_left - 40;
                    robo_obj->Obj_width = robo_obj->Right_edge - robo_obj->Left_edge;
                    obj_status = TRUE;
                }
            }

            if (left_edge_point.x < 160 && right_edge_point.x > 160)
            {
                robo_obj->LorRTag = 1;
                robo_obj->edge_depth = edge_left>edge_right ?
                                   depth_Right+KRobot_length :
                                   depth_left+KRobot_length;

                robo_obj->ObjBox = box;
                robo_obj->Left_edge = edge_right;
```

```
                robo_obj->Right_edge = edge_left;

                robo_obj->Obj_width = robo_obj->Left_edge + robo_obj->Right_edge;
                obj_status = TRUE;
            }

            if (left_edge_point.x >= 160 && right_edge_point.x > 160)
            {
                if (edge_left < 500 )
                {
                    robo_obj->LorRTag = 0;
                    robo_obj->edge_depth = depth_left + KRobot_length;

                    robo_obj->ObjBox = box;
                    robo_obj->Left_edge = edge_right;
                    robo_obj->Right_edge = edge_left + 40;

                    robo_obj->Obj_width = robo_obj->Left_edge - robo_obj->Right_edge;
                    obj_status = TRUE;
                }
            }
        }
    }

    return obj_status;
}
```

最终的运行结果如图12-6所示。

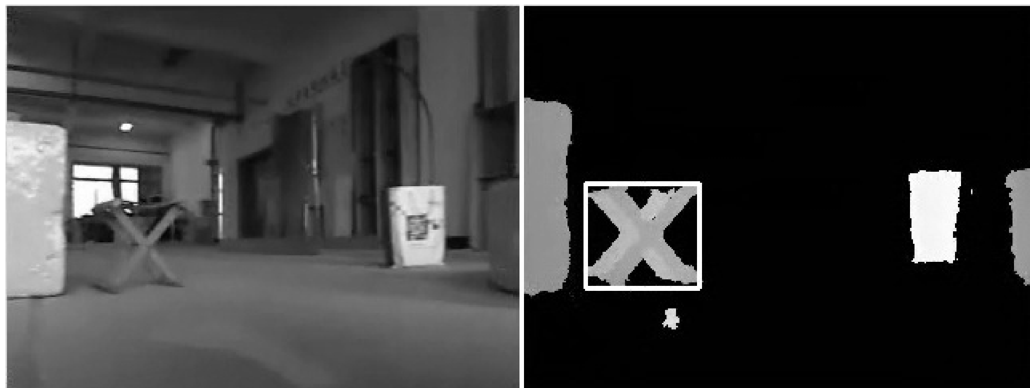


图12-6　障碍物判定结果

从彩色图像中可以看到，在机器人附近有4个物体（在深度图上能够表示出来）。在对应的深度图中，经过障碍物判断规则之后，右边的两个物体由于离机器人较远而没有被识别为障碍物，最左边的那个物体同样不构成威胁。只有左边第二个物体——凳子，对机器人的行进构成了威胁，于是被识别为障碍物。

## 12.4.4　制定机器人避障规则

对物体深度图进行障碍物判别之后，可以得到障碍物数组，该数组记录了机器人左边，中间以及右边的障碍物情况。制定避障规则的目的是为了让机器人在不同的障碍物情况下，执行相应的运动策略。在具体调试过程中，仅考虑两种障碍物的关系，第一种是只有一个障碍物，第二种是存在两个位置的障碍物。由于机器人采用的是全向底盘，运动控制非常简便，仅需要X方向和Z方向的两个距离即可，也就是边缘距离和深度距离。这两种情况的处理策略如下。

- 一个障碍物可能存在三种位置关系，系统将根据此关系给机器人发送相应的边缘距离。比如，当障碍物位于左边时，将右边缘距离发送给机器人。
- 两个障碍物存在着合并的可能性，比如都位于左边，此时机器人会将这两个障碍物视为一个整体，只需要关注两个障碍物中更右边那个的右边缘距离即可。而如果两个障碍物一个在左边，一个在右边，就需要采用两种策略来计算这两个障碍物之间的距离。当该距离大于机器人宽度时，可以直接穿过。而当该距离小于机器人宽度时，就需要分别计算避开左边和右边障碍物的偏移距离，并选择偏移距离最小的避障路线。 由于这部分功能的代码涉及了大量的底层控制逻辑，所以就不在这里列出了。最终的实际避障效果如图12-7所示。

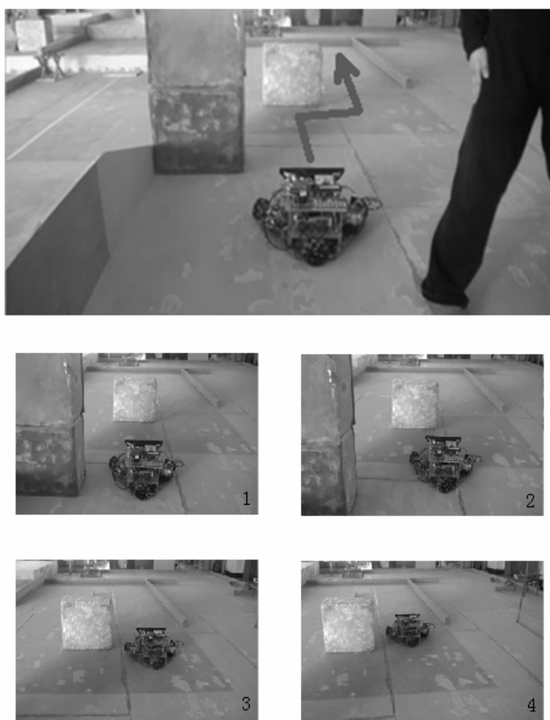



图12-7　避障操作实际效果

可以看出，在面对如图所示的两个障碍物时，机器人准确地判断出这两个障碍物都位于其左边，从而选取了右侧绕过的方法，完成了避障任务。

## 12.5　利用骨架数据实现人体跟踪

Kinect的精华部分在于其强大的人机交互能力，利用Kinect可以很容易地提取出用户的骨架，识别出用户的姿态。本节将介绍机器人通过识别人体骨架来完成人体跟踪的任务。

通过Kinect SDK可以得到人体的20个骨骼点以及该用户的坐标和深度距离。这样通过前面介绍的距离计算方法，就可以计算出用户与机器人的位置偏移情况，从而控制机器人的移动。

值得一提的是，目前的Kinect SDK提供了一种用户追踪模式。在多用户情况下（目前最多可对6个用户进行骨架识别），Kinect提取的每个骨架都有自己唯一的ID，把这个ID写入追踪模式（目前最多支持同时追踪两个用户），可以将相应的状态置位，这就使当前用户不会受到其他用户的干扰。Kinect视角下的跟踪情况如图12-8和图12-9所示。




图12-8　时刻A的跟踪情况




图12-9　时刻B的跟踪情况

人体跟踪功能的逻辑控制代码如下：

```
if (human_command.human_tracking)    //人体跟踪模式启动
{
    EnterCriticalSection(&com_write_criticalsect);
    if (human_command.move_up)    //同时机器人启动
    {

        if ( human_command.human_skeleton )    //同时发现骨架
        {
            switch(human_command.HumanDirection)
            {
            case 1:
                COMM[0] = TrackingMode_Left;    //表示人体跟踪模式
                break;
            case 2:
                COMM[0] = TrackingMode_Right;
                break;
            default:
                COMM[0] = TrackingMode_Left;
            }
            /*向机器人传输人体所在位置，代码略*/
        }
        Else //追踪模式启动之后骨架丢失，令机器人后退
        {
            /*向传输停止指令，代码略*/;
        }
    }
    LeaveCriticalSection(&com_write_criticalsect);
}
```

这样就保证了只有在开启跟踪模式且骨架存在的情况下，机器人才会跟踪人体，跟踪方式为直线移动。而当骨架丢失时，机器人会立即停止，以防意外发生。通过Kinect获取人体骨架稳定性好、耗时少，基本可以保证机器人始终跟随用户。对应于图12-8和图12-9的两个时刻，机器人的实际运行状态如图12-10和图12-11所示。



图12-10　时刻A的实际运行状态



图12-11　时刻B的实际运行状态

## 12.6　利用麦克风进行声音定位

我们可以利用Kinect的麦克风阵列实现很多语音应用，比如本项目中非常实用的声音定位功能。用户在离开机器人的视野范围后，可以通过语音来控制机器人，此时机器人会根据Kinect获取的声音来判断用户的位置，然后进行自转，使自己始终面向用户。

尽管Kinect SDK包含了丰富的声音处理示例，但在1.0版本中，基于C++语言的示例并不支持语音识别功能。本项目中的声音定位程序参考了Kinect SDK提供的MicArrayEchoCancellation示例的部分代码，通过声音获取选择角度，从而实现定位功能。

理论上，Kinect的麦克风阵列以线阵方式排列，探测范围能够达到180°（–90° ~ 90°），但在实际测试中，只有在–40° ~ 40°的角度范围内，声音效果才比较明显。考虑到Kinect麦克风阵列的这一特点，为了应对复杂的应用场景，我们采取了如下控制策略。

(1) 由于Kinect麦克风阵列测试具备上述特点，因此机器人视野内是否存在骨架就成为了判断音源位置（来自机器人前方或后方）的依据。这里假设Kinect提取骨架的视野角度范围是–30° ~ 30°。

(2) 受环境影响，机器人可能无法一次自传到位，因此需要用户不断地通过声音来修正其位置。

(3) 如果机器人视野内没有骨架，就表示接收到的声音来自后方，但要排除骨架处于其视野盲区的可能性（即角度$a$的绝对值大于30°）。此时，机器人的自转角度为$(180-a)°$，方向由角度正负决定（正角度表示机器人前进方向的左侧）。其他情况下，机器人只自转90°。

(4) 如果机器人视野内有骨架，则可以直接根据角度自转。

其中采集声源方向的代码与SDK示例一致，这里不再赘述，仅列出声控的逻辑代码以供参考：

```
if ( human_command.human_voice )
{
```

```
    EnterCriticalSection(&com_write_criticalsect);
    COMM[1] = COMM[2] = 0x00;
    COMM[3] = COMM[4] = 0x00;

    //如果角度为正，说明用户在左侧，此时判断有无骨架
    if ( human_command.voice_angle > 0.0 )
    {
        //如果没有骨架，说明用户在后方，需要自转180°，假设骨架视角只有30°
        if ( !human_command.human_skeleton )
        {
            //若角度小于30度，用户肯定在机器人后方
            if ( human_command.voice_angle <= 30.0 )
            {
                /*向机器人发送指令，自转180°，代码略*/
            }
            //角度大于40°，只自转90°
            else if(human_command.voice_angle >= 40.0 )
            {
                /*向机器人发送指令，自转90°，代码略*/
            }
            else //否则人在前面，只是看不到而已，即30°<angle<40°
            {
                /*向机器人发送指令，转至voice_angle角度，代码略*/
            }
        }
        //有骨架，直接根据角度自转
        else
        {
            /*向机器人发送指令，转至voice_angle角度，代码略*/
        }

    }

    //此时声音在右侧，需要通过骨架判断是前方还是后方
    if ( human_command.voice_angle < 0.0 )
    {
        /*同左侧，代码略*/
    }

    if ( human_command.voice_angle == 0.0 )
    {
        /*向机器人发送指令，保持静止，代码略*/
    }
    LeaveCriticalSection(&com_write_criticalsect);
}
```

这些策略仅仅是折中的方案，因为实际操作者和Kinect提取的用户骨架可能不是同一个人。不难看出，声音定位极易受到外界环境的影响，误判率较高，因此需要通过语音识别功能辅助判断，这也是进一步改进的重点。实际效果如下图所示，图12-12为初始状态，图12-13中用户向右移动并发出声音，机器人定位后自转面向用户。



图12-12　初始状态



图12-13　声源跟踪结果

## 12.7　完善人机交互演示系统

设计出一套适合该应用的统一自然交互系统，可以极大地提高机器人的可操作性。想要让机器人走进家庭，就需要让它更多地和人进行交流，而如果采用传统的外设来控制，用户体验性和适用性就会大打折扣。机器人如果能够读懂人的肢体语言，用户就能非常方便地对其进行控制，并减少操作的麻烦。幸运的是，利用Kinect完全可以实现此功能。

总体来讲，机器人具有人体跟踪、行走避障以及声音定位三个功能，可以通过一整套动作来实现不同功能之间的切换。经过大量实验，该项目采用以下姿态来完成整个演示系统，如图12-14和图12-15所示。




伸开双臂：机器人进入人体跟踪模式




伸直右臂：机器人进入行走避障模式




伸直左臂：机器人进入声音定位模式

图12-14　机器人各个模式之间的切换




左手向前：停车指令，机器人停止运动




右手向前：启动指令，机器人开始运动

图12-15　控制机器人运动的命令

由于采用姿态控制状态切换，因此设计时要考虑到以下两点。

(1) 由于机器人和用户可能会同时处于运动状态，而人在走动时动作是最放松的，因此控制切换的姿态必须非常特殊。另外，还需要添加延时功能，即当机器人采集到用户的控制姿态后，不会马上响应，而是统计一定时间内该姿态是否持续出现，这样可以防止用户的误操作。考虑到安全因素，这里并没有对停车指令进行延时判断，以确保该动作能够立即得到响应。

(2) 利用SDK提供的骨架跟踪机制可以较好地解决多用户操作问题。由于骨架具有唯一的ID，因此当某个用户利用姿态向机器人发出命令后，机器人将只接受该骨架ID的姿态命令，不会受到其他用户的干扰，除非该用户离开机器人的视野。这是因为目前的SDK骨架追踪机制只在骨架可见的情况下生效，暂不支持骨架记忆识别。在调试中，机器人视野内有两名用户，当一个用户完成了定义的操作（伸开双臂）后，即可获得机器人的控制权，机器人将不再响应其他用户的命令，这样就实现了机器人对骨架的跟踪功能。当然，如果其他用户距离机器人过近，会影响控制用户的骨架，这种状况下可能会出现误操作。

## 12.8　小结

至此，关于KRobot项目的各部分技术实现就介绍完了。在这一章中，我们了解了Kinect的创新用法，即将其作为机器人的“眼睛”和“耳朵”，使机器人能够观察并识别外界环境。与前面提到的几种应用不同，本项目不仅要响应视觉主体（用户），还需要分析其他物体，这本身就是相当复杂的工作。不仅如此，KRobot还很好地将Kinect的特性融入了机器人的操作：利用骨骼追踪功能控制机器人追踪人体；利用声音识别功能完成声源追踪；利用自然交互功能完成对机器人的便捷操控。这些都是值得读者学习和借鉴的。相信在不久的将来，会有越来越多的机器人项目以Kinect作为传感器。这将使机器人的识别能力和交互能力得到极大的提高，让它更像真正的“人”。

附录 A

# Kinect for Windows SDK类、结构类型和枚举类型

表A-1将列出Kinect for Windows SDK中使用到的所有类，并对它们做了简单的功能描述。

表A-1　Kinect for Windows SDK类

| 名　　称 | 描　　述 |
|---|---|
| AllFramesReadyEventArgs | 全数据流回调事件 |
| BeamAngleChangedEventArgs | 俯仰角更改事件 |
| BoneOrientation | 骨骼点旋转信息 |
| BoneOrientationCollection | 骨骼点旋转信息列表 |
| BoneRotation | 旋转参数 |
| ColorImageFrame | 彩色数据帧 |
| ColorImageFrameReadyEventArgs | 彩色数据流回调事件 |
| ColorImageStream | 彩色数据流 |
| DepthImageFrame | 深度数据帧 |
| DepthImageFrameReadyEventArgs | 深度数据流回调事件 |
| DepthImageStream | 深度数据流 |
| ImageFrame | 图像数据帧 |
| ImageStream | 图像数据流 |
| JointCollection | 骨骼点数据列表 |
| KinectAudioSource | Kinect声音数据流 |
| KinectSensor | Kinect运行时 |
| KinectSensorCollection | Kinect运行时列表 |
| Skeleton | 单一骨骼数据 |
| SkeletonFrame | 骨骼数据帧 |
| SkeletonFrameReadyEventArgs | 骨骼跟踪数据流回调事件 |
| SkeletonStream | 骨骼数据流 |
| SoundSourceAngleChangedEventArgs | 声源角度改变回调事件 |
| StatusChangedEventArgs | Kinect传感器连接状态改变回调事件 |

表A-2将列出Kinect for Windows SDK中使用到的所有结构类型，并对它们做了简单的功能描述。

表A-2　Kinect for Windows SDK结构类型

| 名　　称 | 功能描述 |
|---|---|
| ColorImagePoint | 彩色图的单一像素 |
| DepthImagePoint | 深度图的单一像素 |
| Joint | 骨骼数据中单一骨骼点 |
| Matrix4 | 4x4矩阵 |
| SkeletonPoint | Kinect空间下的一个骨骼点位置 |
| TransformSmoothParameters | 骨骼点光滑处理参数 |
| Vector4 | 四维向量 |

表A-3将列出Kinect for Windows SDK中使用到的所有枚举类型，并对它们做了简单的功能描述。

表A-3　Kinect for Windows SDK枚举类型

| 名　　称 | 功能描述 |
|---|---|
| BeamAngleMode | 俯、仰角设置方式 |
| ColorImageFormat | 彩色图片格式 |
| DepthImageFormat | 深度图片格式 |
| DepthRange | 深度数据取景模式 |
| EchoCancellationMode | 回声消除选项 |
| FrameEdges | 骨骼数据相对深度图边界状态 |
| JointTrackingState | 骨骼点跟踪状态 |
| JointType | 骨骼点类型 |
| KinectStatus | Kinect传感器状态 |
| SkeletonTrackingMode | 骨骼跟踪模式 |
| SkeletonTrackingState | 骨骼跟踪状态 |

## 图灵原创

cocos2d-x手机游戏开发：跨iOS、Android和沃Phone平台

论道HTML5

Go语言•云动力

推荐系统实践

Unity 3D游戏开发

大道至易：实践者的思想

思考的乐趣：Matrix67数学笔记

Node.js开发指南

Go语言编程

DBA的思想天空——感悟Oracle数据库本质

Kinect人机交互开发实践

# KINECT 人机交互开发实践

除了为读者介绍Kinect开发的相关知识外，书中通过大量篇幅分析了一些精挑细选的实际项目，这些项目给开发者提供了很好的开发方向和设计思路。从应用的整体设计思路到具体的算法实现，每个项目都给出了实际开发中需要注意的地方，这值得开发者花精力去了解。

**——邹欣，微软首席研发经理，《编程之美》作者**

可以说，Kinect的诞生预示着一个全新的计算机应用领域的开拓。本书将读者群定位在具有极大创造力和实现能力的个人和企业开发者身上，详细介绍了Kinect for Windows开发的方方面面，是有志投身该领域的开发者不可错失的一本书。

**——马宁，MVP，OpenXLive CTO**

前些日子刚刚看到国内能够买到Kinect for Windows，感觉通过Kinect来控制电脑的革命性人机交互方式离我们的实际生活更近了。遍寻各大网上书店，没有相关的中文开发资料，深感遗憾。本书的推出填补了国内Kinect for Windows开发领域的空白。幸运的是，我成为了本书的第一批读者，原以为它可能就是官方SDK文档的梳理，但随着阅读的深入，本书带给了我极大的惊喜。它不仅仅停留在概念的阐述、基本功能的实现，更有吸引力的是它讲述了从构思、设计到实现一个Kinect交互应用的完整过程，同时又对Kinect应用的用户友好性提出了真知灼见，极具参考价值。

**——姜泳涛，MVP，TechEd讲师**

这是一本很适合体感开发初学者阅读的书。从Kinect for Windows开发环境的搭建，再到Kinect彩色图像数据、深度图像数据、骨骼追踪数据、音频数据的获取与使用，以及人脸识别等，讲解非常细致，知识体系非常完整。这本书是作者大量实践经验的结晶，相信对Kinect for Windows体感开发者会有很大的帮助，强烈推荐！

**——王峰，cnKinect.com创始人**

封面设计：PLATO·创意机构
图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186转604

分类建议 计算机/人机交互/Kinect

人民邮电出版社网址：www.ptpress.com.cn




ISBN 978-7-115-30029-4
定价：39.00元